Infrared Contracts Security Review

Auditors

Noah Marconi, Lead Security Researcher

0xRajeev, Lead Security Researcher

Cryptara, Security Researcher

Chinmay Farkya, Associate Security Researcher

Report prepared by: Lucas Goiriz

Report date: January 20, 2025

Contents

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

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

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 28 days in total, Infrared Finance engaged with Spearbit to review the infrared-contracts protocol. In this period of time a total of **75** issues were found.

Summary

Project Name	Infrared Finance
Repository	infrared-contracts
Commit	33bc49c3
Type of Project	DeFi, Staking
Audit Timeline	Dec 13th to Jan 10th
Fix period	xxx xxx

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	2	2	0
Medium Risk	8	5	2
Low Risk	17	7	7
Gas Optimizations	10	0	0
Informational	37	17	3
Total	**75**	**32**	**12**

5 Findings

5.1 Critical Risk

5.1.1 Missing `_grantRole` for `GOVERNANCE_ROLE` will prevent calling of `onlyGovernor` functions including upgrades

Severity: Critical Risk

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

Summary: Missing the granting of role for `GOVERNANCE_ROLE` to any privileged address will prevent calling of `onlyGovernor` functions including upgrades across `InfraredBERA` contracts along with `Voter` and `BribeCollector` core contracts of Infrared.

Finding Description: All `InfraredBERA` related contracts derive from `UUPSUpgradeable` and `AccessControlUpgradeable` where `_authorizeUpgrade` is restricted to the `GOVERNANCE_ROLE`. `InfraredBERA` also has several protocol-critical `onlyGovernor` functions of `setWithdrawalsEnabled()`, `setFeeShareholders()`, `setDepositSignature()`, `setDepositor()`, `setWithdrawor()`, `setClaimor()`, `setReceivor()`.

However, the `GOVERNANCE_ROLE` is not granted to any privileged address during initialization, as done in `Infrared.sol` via `_grantRole(GOVERNANCE_ROLE, _admin)`.

This is the case in `Voter.sol`, which prevents upgrading it and calling `onlyGovernor` functions of `setMaxVotingNum()`, `whitelistNFT()`, `killBribeVault()` and `reviveBribeVault()`. This is also the case in `BribeCollector.sol`, which prevents upgrading it and calling `onlyGovernor` function of `setPayoutAmount()`.

Related unit tests succeed because this role is granted during their setup.

Impact Explanation: High, because several protocol-critical functions can never be called, for example:

1. `setWithdrawalsEnabled()` cannot be called to enable voluntary withdrawals in future.
2. `setDepositSignature()` cannot be called to set valid deposit signatures for validators, which will prevent `InfraredBERADepositor` deposits from executing and effectively preventing liquid staking to function.
3. Voter bribe vaults cannot be killed or revived.
4. `BribeCollector.setPayoutAmount()` cannot be called to set `payoutAmount`, which allows anyone to claim all bribe tokens for free (because default `payoutAmount` is 0).
5. None of these contracts can be upgraded in future.

Likelihood Explanation: High, because `_grantRole` for `GOVERNANCE_ROLE` is missing in all relevant `initialize()` functions and also absent in related deployment scripts.

Recommendation: Consider adding something similar to `_grantRole(GOVERNANCE_ROLE, _admin)` in all relevant `initialize()` functions.

Infrared: Fixed in PR 283, PR 294 and PR 316.

Spearbit: Reviewed that:

1. PR 283 fixes the issue as recommended using `_grantRole(GOVERNANCE_ROLE, _gov)` for `InfraredBERA` contracts.
2. PR 294 fixes the issue as recommended using `_grantRole(GOVERNANCE_ROLE, _gov)` for `BribeCollector`.
3. PR 316 fixes the issue as recommended using `_grantRole(GOVERNANCE_ROLE, _gov)` for `Voter`.

5.2 High Risk

5.2.1 Missing `_grantRole` for `KEEPER_ROLE` will prevent calling of critical Keeper functions

Severity: High Risk

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

Summary: Missing the granting of role for `KEEPER_ROLE` to any privileged address will prevent calling of Keeper functions across `InfraredBERA` contracts and Voter core contracts of Infrared.

Finding Description: The `KEEPER_ROLE` in `InfraredBERA` contracts is the only privileged address that is allowed to execute deposits until the expiry of `FORCED_MIN_DELAY == 7 days` after which anyone is allowed to force stake deposits. This will similarly apply to voluntary withdrawals once they are enabled. Also, Keeper is the only entity allowed to `sweep()` forced withdrawals.

Keeper is also the only privileged address that is allowed to call `createBribeVault()` for creating new bribe vaults for staking tokens in the Infrared Voting contract.

However, the `KEEPER_ROLE` is not granted to any privileged address during initialization, as done in `Infrared.sol` via `_grantRole(KEEPER_ROLE, _admin)`. Related unit tests succeed because this role is granted during their setup.

Impact Explanation: Medium, because without a Keeper executing staking deposits, users are forced to wait until the expiry of `FORCED_MIN_DELAY == 7 days` to force stake deposits. This delay will lead to a loss of any accrued rewards that users may have obtained if their staking funds had immediately been deposited towards validators.

Medium, because without a Keeper creating new bribe vaults, the RED token-based bribing+voting towards queuing cutting boards for BGT reward emissions from validators will not work. This is a critical goal of Infrared to create a decentralized RED token-based programmatic bribing alternative to the Native Berachain POL-based economic bribing for validator emissions. Without this, Infrared will fallback to the Native Berachain POL bribing mechanism.

Likelihood Explanation: High, because `_grantRole` for `KEEPER_ROLE` is missing in all relevant `initialize()` functions and also absent in related deployment scripts.

Recommendation: Consider adding something similar to `_grantRole(KEEPER_ROLE, _admin)` in all relevant `initialize()` functions.

Infrared: Fixed in PR 283 and PR 316.

Spearbit: Reviewed that:

1. PR 283 fixes the issue as recommended using `_grantRole(KEEPER_ROLE, _keeper)` for `InfraredBERA` contracts.
2. PR 316 fixes the issue as recommended using `_grantRole(KEEPER_ROLE, _keeper)` for `Voter`.

5.2.2 Berachain POL bribes to Infrared can never be claimed

Severity: High Risk

Context: BribeCollector.sol#L81-L83, RewardsLib.sol#L242-L252

Summary: Berachain POL bribes to Infrared are stuck in `BribeCollector` and can never be claimed for Payout token in the auction because `claimFees()` will always revert.

Finding Description: Berachain POL native bribes paid to Infrared validators are harvested to `BribeCollector` periodically with calls to `harvestBribes()`. The expectation is that these are auctioned for a Payout token which then is distributed among Infrared validators. The auction is carried out via `claimFees()` where the winner pays `payoutAmount` of `payoutToken` (WBERA) in return for all the bribe/fee tokens collected so far in `BribeCollector`.

When `claimFees()` makes the call to `infrared.collectBribes(payoutToken, payoutAmount)`, the logic in `collectBribesInWBERA()` is expected to transfer-in the WBERA, convert a part of that to BERA for sending to `InfraredBERA.receivor` and transfer the rest to `ibgtVault`. However, WBERA is never redeemed for BERA in this flow. This causes the BERA transfer to `InfraredBERA.receivor` via `rec.call{value: amtInfraredBERA}` to always revert with `ETHTransferFailed`.

Note: This issue was discovered during a discussion with the client team.

Impact Explanation: Medium, because Berachain POL bribes to Infrared can never be claimed from the protocol and this results in loss for the Infrared validators and stakers of `ibgtVault`.

Likelihood Explanation: High, because this happens unconditionally any time `claimFees()` is called for claiming Berachain POL bribes.

Recommendation: Consider adding a call to `IWBERA(wbera).withdraw(amtInfraredBERA);` for redeeming WBERA to BERA before the call to IBERA receivor.

Infrared: Fixed in PR 243.

Spearbit: Reviewed that PR 243 fixes the issue as recommended.

5.3 Medium Risk

5.3.1 Validator cutting board weights can disconnect from onchain voting outcomes

Severity: Medium Risk

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

Summary: Validator cutting board weights can disconnect from onchain voting outcomes leading to losses for specific BGT receivers and reputational damage.

Finding 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 RED onchain voting distribution outcomes as actual cutting board weights for each Infrared validator.

However, the provided `_weights` data and `_startBlock` in this function 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 Explanation: 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 Explanation: 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.

Infrared: Acknowledged.

Spearbit: Acknowledged.

5.3.2 Missing call to `sweep()` before calling `collect()` in `harvestOperatorRewards()` may result in inaccurate rewards

Severity: Medium Risk

Context: RewardsLib.sol#L307-L313

Summary: Missing sweep before calling collect in `harvestOperatorRewards()` may result in inaccurate rewards because of stale data being used.

Finding Description: In `harvestOperatorRewards()`, there is a potential issue with stale data being used when calculating fees via `collect()`. Specifically, `InfraredBERA.collect()` relies on the `shareholderFees` value, which may not be up-to-date unless a `sweep()` is performed beforehand. While `mint()` triggers a `sweep`, `collect()` does not. This discrepancy means that when `harvestOperatorRewards()` is called, it may process rewards using an outdated `shareholderFees` value, potentially leading to incorrect fee calculations and distribution.

Impact Explanation: Medium, because this can result in inaccurate rewards allocation due to outdated fee data, causing mismanagement of protocol fees and discrepancies in the operator rewards distribution, which could negatively affect the trust and reliability of the reward system.

Likelihood Explanation: Medium, because `harvestOperatorRewards()` is expected to be invoked regularly in the protocol to manage operator rewards. If a `sweep()` is not explicitly called prior to `collect()`, there is a consistent risk of stale data usage.

Recommendation: To ensure accurate fee calculations, `harvestOperatorRewards()` should explicitly trigger a `sweep()` operation in `InfraredBERA` before calling `collect()`. This will ensure that `shareholderFees` is always updated to the latest value before rewards are processed. This adjustment will align the behavior of `harvestOperatorRewards` with the safeguards provided by `mint`, reducing the risk of incorrect fee allocation.

Infrared: Fixed in PR 326.

Spearbit: Reviewed that PR 326 fixes the issue as recommended by adding a call to `compound()` (which calls `sweep()`) in `harvestOperatorRewards()` before calling `collect()`.

5.3.3 A malicious reward token can DoS reward claiming for all users in a vault

Severity: Medium Risk

Context: MultiRewards.sol#L231-L240

Summary: A malicious reward token can DoS reward claims for all users in a vault to which it is added. This will affect the claiming process for all the reward tokens because they are transferred in a loop, which can be forced to revert.

Finding Description: In `getRewardForUser()`, after the rewards state is updated, the reward tokens earned by the user are transferred to them iteratively in a loop.

However, if any of these reward tokens become malicious (or is paused, where pausing is controlled by external token governance), then the reward claiming process can be forced to revert for all users in that particular `InfraredVault`.

While there is a way to remove whitelisting of a reward token from the Infrared system by infrared governance, there is no way to remove a reward token from a vault once it's added to it. In an `InfraredVault`, governance can only add more reward tokens but not remove any. The entire `rewardTokens` list is always processed in a `getRewardForUser()` call.

Impact Explanation: High, because the reward claim process cannot succeed for any tokens and any users in an infrared vault with a malicious reward token.

Likelihood Explanation: Low, because the tokens are whitelisted by infrared governance. This problem can occur even if one of the reward tokens is paused, for example, by the external token's governance.

Recommendation: Consider using one or more of the following:

1. Logic in the reward transfer loop to skip past failing token transfers.
2. A function for the governance to remove reward tokens from the `rewardTokens` list of an `InfraredVault`.
3. Nomad's ExcessivelySafeCall.sol to prevent return bomb attack.
4. A stringent whitelist process to only add well-known tokens.

Infrared: Fixed in PR 401.

Spearbit: Reviewed that PR 401 adopts recommendation (1) by using a low-level `call` with `gas: 200000`.

5.3.4 Precision loss in reward calculation leads to incorrect accounting for residual amounts

Severity: Medium Risk

Context: MultiRewards.sol#L285-L308

Summary: The precision loss in reward calculation leads to incorrect accounting for residual amounts.

Finding Description: In the previous review, the issue of precision loss in the calculation of reward rate was pointed out. Accounting for "residual amounts" was introduced as a fix.

When `reward` amount is divided by `rewardsDuration`, any remainder is stored as `rewardResidual` and utilized in later reward distributions so that it does not get lost. However, the problem still exists in cases where there is a leftover amount from the current running distribution period.

We see that the residual amount `reward / rewardsDuration` is correctly stored as `rewardResidual`. When there is a `leftover` amount, it is added into the `rewardRate` calculation. However, the precision loss there is not considered. Any truncation due to the division `leftover / rewardsDuration` is not accounted for in `rewardRate` and neither in the `rewardResidual`. This can lead to a loss of that reward amount.

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

Likelihood Explanation: High, because the irrecoverable rewards keep accumulating on every invocation of `_notifyRewardAmount()` for every `rewardsToken`.

Recommendation: The calculation needs to be reconsidered with the amount lost to division in `leftover / rewardsDuration` also being added to `rewardResidual`.

Infrared: Fixed in PR 398.

Spearbit: Reviewed that PR 398 resolves the issue as recommended.

5.3.5 Incomplete `InfraredKeeperScript` may lead to unexpected Keeper behavior

Severity: Medium Risk

Context: InfraredKeeperScript.s.sol#L11-L63

Summary: Incomplete `InfraredKeeperScript` with partial functionality and hardcoded testnet addresses may lead to unexpected Keeper behavior if they are not completed/fixed/validated properly.

Finding Description: Keepers have a special role in the Infrared protocol to automate the calling of several important functions in a timely manner. Such functions include `queueNewCuttingBoard()`, `queueBoosts()`, `cancelBoosts()`, `queueDropBoosts()`, `cancelDropBoosts()`, `createBribeVault()`, `registerVault()`, `harvestBase()`, `harvestVault()` and `harvestBribes()`.

However, `InfraredKeeperScript` which is intended to capture all such batched keeper jobs is incomplete (with commented code, todo's) and currently has hardcoded testnet addresses.

Impact Explanation: High, because the incomplete/incorrect calling of important Infrared functions will lead to unexpected protocol behavior.

Likelihood Explanation: Low, because this is expected to be completed/corrected before mainnet deployment.

Recommendation: Complete and correct all the expected batched keeper jobs and contract addresses with appropriate validation before mainnet deployment.

Infrared: Work in progress: PR 286.

5.3.6 Anyone can cause DoS to the updating of a vault's `rewardsDuration`

Severity: Medium Risk

Context: MultiRewards.sol#L345-L352

Summary: The check in `MultiRewards.sol::_setRewardsDuration()` requires that an InfraredVault's `rewardsDuration` can only be updated after the last running reward period has ended. But anyone can cause DoS to such a call to prevent the `rewardsDuration` from being updated for a vault.

Finding Description: The following is the flow for updating the rewards duration for a `rewardToken` on a specific vault: `Infrared::updateRewardsDurationForVault()` ⇒ `VaultManagerLib::updateRewardsDurationForVault()` ⇒ `InfraredVault::updateRewardsDuration()` ⇒ `MultiRewards::_setRewardsDuration()`.

The `_setRewardsDuration()` finally checks that the `block.timestamp > periodFinish`, otherwise it reverts.

Because `Infrared.addIncentives()` is a public function, anyone can come up with a small amount : even 1 wei of a `rewardToken` and the vault will be notified with 1 wei of a reward and advance the `periodFinish` value because of the logic in `MultiRewards::_notifyRewardsAmount()`.

The call flow is `Infrared::addIncentives()` ⇒ `VaultManagerLib::addIncentives()` ⇒ `InfraredVault::notifyRewardAmount()` ⇒ `MultiRewards::_notifyRewardAmount()`.

The calculation there does not check if the resultant `rewardRate` is 0$\Rightarrow$it just advances the `periodFinish` value forward. This means anyone can notify a reward amount of even 1 wei for a reward token on a vault, which would lead to an increase in the `periodFinish` timestamp.

This can be used to cause DoS to `updateRewardsDurationForVault()` call forever, and it would become impossible to update the `rewardsDuration` for any `rewardToken` on any vault if an attacker kept repeatedly advancing the `periodFinish` value.

Additionally, given that none of the functions in `addIncentives()` ... ⇒ ... ⇒ `notifyRewardAmount()` flow have the `whenNotPaused` modifier, we cannot even pause the vault to perform this update operation.

The same thing can happen with normal usage via `harvestVault()` calls because those too notify new rewards to a vault and advance the `periodFinish` timestamp forward.

Impact Explanation: Low, because the `rewardsDuration` is not meant to be regularly updated.

Likelihood Explanation: High, because this will always happen via `harvestVault()` calls during normal usage and can be easily used as an attack preventing the update of `rewardsDuration` for any `rewardToken` on any `InfraredVault`.

Recommendation: Consider one of the following:

1. This will require a refactoring of the `_setRewardsDuration()` logic including :
    - Recalculate the leftover amount from the currently running reward period.
    - Calculate the new reward rate using the leftover amount and the new `rewardsDuration` value.
    - Start a new reward period with the above parameters.
    - Remove the `block.timestamp > periodFinish` check.
2. Consider adding `whenNotPaused` modifier to one of the functions in `addIncentives()` ... ⇒ ... ⇒ `notifyRewardAmount()` flow so that we can pause the vault to perform this update operation.

Infrared: Fixed in PR 376.

Spearbit: Reviewed that PR 376 fixes the issue using the recommended `whenNotPaused` modifier.

5.3.7 Berachain Consensus Layer implementation and specification not being in-sync and in flux is risky

Severity: Medium Risk

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

Summary: Berachain Consensus Layer implementation and BeaconKit specification not being in-sync and in flux is risky for Infrared liquid staking.

Finding Description: Berachain Consensus Layer is based on BeaconKit, which is a modular framework for building EVM based consensus clients. Its status says: "*This project is work in progress and subject to frequent changes as we are still working on wiring up the final system. Audits on BeaconKit are still ongoing, and in progress at the moment. We don't recommend using BeaconKit in a production environment yet.*"

However, it is not clear which aspects of BeaconKit specification will actually be part of its Berachain implementation. For example, the below two aspects, are mentioned in its README but do not appear to be enforced yet:

1. "*`EffectiveBalance` is capped at `MaxEffectiveBalance`. Any Balance in excess of `MaxEffectiveBalance` is automatically withdrawn.*"

2. "*The funds deposited will be locked and the validator will stay inactive until the a minimum staking balance is hit. The minimum staking balance is set to* `EjectionBalance + EffectiveBalanceIncrement` *to account for hysteresys.*"

Additionally, `InfraredBERADepositor.execute()` enforces a deposit amount of `INITIAL_DEPOSIT` for the first deposit to any Infrared validator. `INITIAL_DEPOSIT` is hardcoded to a value of `32 ether` in `InfraredBERAConstants`. However, the value of `BeaconDeposit.MIN_DEPOSIT_AMOUNT_IN_GWEI` was changed from 32 BERA to 10K BERA equivalent in Berachain's recently merged PR 549, which will cause all initial deposits from Infrared BERA liquid staking to always revert with `InsufficientDeposit` if `InfraredBERAConstants` are not updated.

Impact Explanation: High, because any disconnect/deviation between Infrared liquid staking and Berachain Consensus Layer implementation/specification will result in unexpected behavior such as failed/stuck deposits.

Likelihood Explanation: Low, because it is expected that Infrared protocol will closely follow the Berachain Consensus Layer implementation and specification so that they are validated to be in-sync before mainnet deployment.

Recommendation:

1. Ensure that Infrared liquid staking is validated to be in-sync with the final Berachain Consensus Layer implementation/specification and configuration before mainnet deployment.
2. Ensure that any findings from the ongoing Beacon Kit security contest are evaluated for impact to Infrared's interactions.
3. Consider:
    - Converting constants in `InfraredBERAConstants` to protocol state variables controlled by governance so that they may be changed appropriately if/when required in future.
    - Enforcing the `INITIAL_DEPOSIT` requirement for all deposits in `InfraredBERADepositor.execute()` because that is enforced in `BeaconDeposit.deposit()` on all deposits and not only the first one.
    - Changing the name of `INITIAL_DEPOSIT` to `DEPOSIT_AMOUNT` to match the enforced check.

Infrared: Acknowledged. Possible work in progress.

Spearbit: Acknowledged.

5.3.8 Rewards harvesting in vaults will be blocked if the RED token is paused

Severity: Medium Risk

Context: RewardsLib.sol#L176-L180

Summary: All Infrared vaults have a mechanism to harvest rewards from BGT emissions whenever someone tries to claim their rewards, or can be called directly from Infrared contract. But these harvest calls will be blocked if the RED token is ever paused.

Finding Description: The reward claim process calls `RED.mint()` in `harvestVault()` function. The flow is: `getRewardForUser()` ⇒ `onReward()` ⇒ `harvestVault()` ⇒ `RED.mint()`.

Whenever there are new BGT rewards (which means some BGT emissions have accrued and claimed from Bera reward vaults), the `harvestVault()` function tries to mint a proportional amount of RED tokens.

However, token transfers (including mints, burns etc.) can be paused on the RED token. So if RED mint fails, the reward harvesting calls will revert.

Impact Explanation: High, because this will make all `getRewardForUser()` calls revert, i.e. this will prevent harvesting of new rewards.

Likelihood Explanation: Low, because this will only happen if the RED token is paused. And the RED token will only be paused in very rare situations.

Recommendation: Either adopt a try/catch approach when minting RED tokens during `harvestVault()`, or exclude mint operations by modifying the `_update()` function in `ERC20PresetMinterPauser.sol`.

Infrared: Fixed in PR 378.

Spearbit: Reviewed that PR 378 resolves the issue as recommended by using a try/catch block.

5.4 Low Risk

5.4.1 Incorrect accounting of `rebalancing` may lead to temporary DoS of withdrawals

Severity: Low Risk

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

Summary: Incorrect accounting of `rebalancing` amount in `pending()` may lead to temporary DoS of withdrawals while queuing.

Finding Description: `rebalancing` is meant to track the amount of BERA used for internally rebalancing amongst Infrared validators. This is incremented when withdrawals are queued by the Keeper and decremented when the withdrawals are actually processed to `InfraredBERADepositor` for immediate deposting. This `rebalancing` is used in `pending()` along `InfraredBERADepositor(depositor).reserves()` to estimate "*Pending deposits yet to be forwarded to CL; The amount of BERA yet to be deposited to CL*".

However, this is incorrect because `rebalancing` is incremented while queuing that amount for withdrawal from CL even before withdrawal. This undercounts `InfraredBERA.confirmed()` until the rebalancing amount is actually withdrawn from CL and queued for deposit. Also, once the `rebalancing` amount is withdrawn and processed to queue it up for deposit, it is already part of the `depositor.reserves()`. So it looks like the `rebalancing` variable accounting may not be necessary at all.

Impact Explanation: Low, because `amount > IInfraredBERA(InfraredBERA).confirmed()` may revert in `InfraredBERAWithdrawor.queue()` leading to temporary DoS of withdrawals.

Likelihood Explanation: Low, because this may only happen in edge cases depending on the timing of rebalancing and withdrawals.

Recommendation: Reconsider the use and accounting of `rebalancing` variable in related logic once withdrawals are enabled.

Infrared: Fixed in PR 319.

Spearbit: Reviewed that PR 319 removes the use of `rebalancing` variable entirely. *Note*: Voluntary withdrawals are currently disabled both in Berachain and Infrared. Only the `sweep()` logic in `InfraredBERAWithdrawor` was in scope of this review. `InfraredBERA.burn()` and rest of the related withdraw logic in `InfraredBERAWithdrawor` are outside the scope. There are changes expected to be made via contract upgrades when voluntary withdrawals are enabled.

5.4.2 Absence of inactivity leaks and slashing penalties allows inactive/malicious validators

Severity: Low Risk

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

Summary: Absence of inactivity leaks and slashing penalties currently in Berachain BeaconKit allows inactive/malicious validators to negatively impact the consensus mechanism.

Finding Description: "*BeaconKit does not currently support voluntary withdrawals, nor slashing or inactivity leaks. Therefore a validator keeps validating indefinitely. The only case in which a validator may be evicted from the validator set (and its funds returned) is when ValidatorSetCap is hit and a validator with greater priority is added (i.e. with larger EffectiveBalance or equal EffectiveBalance and larger PubKey in alphabetical order)*" (see the `validator-lifecycle` section of Berachain's docs).

Impact Explanation: High, because with no inactivity leaks or slashing, Infrared Validators, like with other Berachain validators, can simply deposit enough stake with pubkey values high enough balance to remain in high priority validator set but perform no validation to cause DoS or maliciously propose/validate to degrade the CometBFT consensus. There is currently no mechanism to force-exit such a misbehaving validator at the Berachain consensus layer or Infrared protocol.

Likelihood Explanation: Very low, because: (1) validators are currently whitelisted from reputable entities (2) validators lose incentives when they don't produce blocks (3) validators will be required to post collateral to Infrared vaults in future, which will allow Infrared-level inactivity leak and slashing enforcement.

Recommendation: Consider introducing collateral-based Infrared-level inactivity leak and slashing enforcement to complement the current reputation-based whitelisting.

Infrared: Acknowledged.

Spearbit: Acknowledged.

5.4.3 Lack of lower bound check in `setFeeShareholders()` may allow unfair EL yield capture by governance

Severity: Low Risk

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

Summary: Lack of lower bound check in `setFeeShareholders()` may allow governance to accidentally/maliciously set a very low value for `feeShareholders` leading to unfair EL yield capture.

Finding Description: `setFeeShareholders()` allows governance to set the value for `feeShareholders` which is used as the denominator in the fee calculation in `fees = amount / uint256(feeShareholders)`. However, `setFeeShareholders()` does not enforce a lower bound check on `feeShareholders`. For example, if governance sets this value to 1, then the entire Execution Layer (EL) yield accumulated via coinbase priority fees and MEV is captured by governance and nothing is swept to `InfraredBERA` for auto-compounding the principal as intended.

Impact Explanation: Medium, because a significant part of Execution Layer (EL) yield accumulated via coinbase priority fees and MEV is intended to be swept to `InfraredBERA` for auto-compounding the principal. Otherwise, this leads to loss of compounding returns for stakers, which may reduce the amount staked in Infrared validators.

Likelihood Explanation: Low, because governance is expected to only set `feeShareholders` to a reasonable value.

Recommendation: Consider introducing, for example, a `MIN_FEESHAREHOLDERS` value which is set to 4 or 5 i.e. max fee is 25% or 20% respectively. This lower bound check can be enforced in `setFeeShareholders()`.

Infrared: Acknowledged. Checks will be implemented in governance scripts work in progress.

Spearbit: Acknowledged.

5.4.4 Unnecessary setters for staking contracts may cause protocol disruption

Severity: Low Risk

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

Summary: Unnecessary setters for staking contracts may cause protocol disruption if governance accidentally/maliciously updates their addresses.

Finding Description: `InfraredBERA` staking contracts `InfraredBERADepositor`, `InfraredBERAWithdrawor`, `InfraredBERAFeeReceivor` and `InfraredBERAClaimor` are upgradeable proxies, whose implementations can be upgraded if/when required.

However, `InfraredBERA` has `onlyGovernor` setters for changing their addresses from earlier before they were converted into proxies.

Impact Explanation: Medium, because if governance accidentally/maliciously updates their proxy addresses then different parts of the protocol will be disrupted when they start interacting with the new contracts instead of any upgraded implementations of the original proxy. For example, `InfraredBERAWithdrawor.sweep()` calls `IInfraredBERA(InfraredBERA).depositor()).queue()` and if the `depositor` address is changed then the previous `InfraredBERADepositor` state will be lost.

Likelihood Explanation: Low, because governance is not expected to call any of these setters but instead upgrade the proxy implementations if/when required.

Recommendation: Consider removing setters `setDepositor()`, `setWithdrawor()`, `setClaimor()` and `setReceivor()` to prevent calling them.

Infrared: Fixed in PR 315.

Spearbit: Reviewed that PR 315 fixes the issue as recommended.

5.4.5 `InfraredBERAFeeReceivor.collect()` will revert if accumulated shareholder fees is equal to the minimum deposit requirement

Severity: Low Risk

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

Summary: `InfraredBERAFeeReceivor.collect()` will revert if accumulated shareholder fees is equal to the minimum deposit requirement of `InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE`.

Finding Description: `InfraredBERAFeeReceivor.collect()` is called by `InfraredBERA` to collect accumulated shareholder fees. While it enforces the minimum deposit requirement against `min == InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE` against `shf = shareholderFees`, the amount it sends in `IInfraredBERA(InfraredBERA).mint{value: amount}` is `shf - 1`.

However, in the edge case where `shf == min`, check on L81 will pass but `amount < min` and this will cause `mint()` to revert in `_deposit()` where the minimum deposit requirement is enforced again.

Impact Explanation: Low, because this temporarily prevents the collection of accumulated shareholder fees.

Likelihood Explanation: Low, because this is an edge case and the revert will be temporary until the shareholder fees increases.

Recommendation: Consider sending the entire `shf = shareholderFees` in `mint()` instead of `shf - 1`, which appears to be intended to leave behind a dust amount for future gas usage.

Infrared: Fixed in PR 318.

Spearbit: Reviewed that PR 318 fixes the issue as recommended.

5.4.6 A low `payoutAmount` can lead to under-priced BERA bribe sales

Severity: Low Risk

Context: BribeCollector.sol#L77

Summary: A low `payoutAmount` set by the `governor` can lead to under-priced BERA bribe sales.

Finding Description: The `BribeCollector` contract uses an auction-like strategy to convert BERA bribes into the desired `payoutToken`. To do so a `payoutAmount` is set by the `governor` and at any time, all fee (i.e. BERA RewardVault incentive) tokens held in the contract may be swapped for the `payoutAmount`.

If the `payoutAmount` is set too low, `BribeCollector.claimFees` will trade its held fee tokens for under market value.

Impact Explanation: Low, because funds lost to the `ibgtVault` are limited to the difference between the cumulative value of unclaimed incentive tokens in the `BribeCollector` and the `payoutAmount`.

Likelihood Explanation: Low, because the incoming incentive tokens from BERA must worth more than the `payoutAmount` for this issue to occur. Instances where greater than expected amounts, or market values, of tokens entering the contract may occur include:

1. Unexpectedly high incentive rates set on BERA RewardVaults
2. Surges in incentive token value

Recommendation: Exercise caution when setting `payoutAmount`. The amount should be of an equivalent value meaningfully greater than the block rewards from each BERA block. To account for high BERA incentive amounts, or dramatic increases in incentive token value, a `payoutAmount` worth a day's worth of BGT emissions **or greater** would reduce the likelihood of this issue.

Infrared: Acknowledged. We are aware of this dynamic and plan to set the payoutAmount sufficiently high.

Spearbit: Acknowledged.

5.4.7 VeNFT transfers leave unclaimed rewards and do not alter votes

Severity: Low Risk

Context: VotingEscrow.sol#L412

Summary: veNFT transfers leave unclaimed rewards and do not alter votes, which result in lost rewards for the former owner.

Finding Description: The `VoteEscrow` contract permits the transferring of NFT positions. When a veNFT is not locked, it may be transferred.

In `Voter`, votes and rewards are attributed to the `tokenId` and not to the token owner. When a transfer occurs, any existing votes remain in place and any claimable rewards become claimable by the new owner. Transfers without first claiming all fees and bribes, result in lost rewards for the former owner.

Impact Explanation: Medium, because this will result in lost rewards for the former owner.

Likelihood Explanation: Low, because veNFT owners are unlikely to transfer without first claiming all fees and bribes.

Recommendation: Consider:

1. No recommended changes to reset votes. The new owner may do so at their leisure.
2. At minimum, document the behavior and discourage transfers for tokens with unclaimed fees/bribes.
3. Consider token owner checkpointing for reward claiming.

Infrared: Acknowledged.

Spearbit: Acknowledged.

5.4.8 Protocol fees removed from `Infrared` by `recoverERC20()` will lead to inaccurate accounting

Severity: Low Risk

Context: Infrared.sol#L323-L334

Summary: Protocol fees may be removed from `Infrared` by calling `recoverERC20()`, which will lead to inaccurate accounting.

Finding Description: The `governor` has permission to call both `Infrared.recoverERC20()` and `Infrared.claimProtocolFees()`. If a protocol fee token is removed from via `recoverERC20()` the accounting in `$.protocolFeeAmounts[_token]` becomes inaccurate. Doing so has implications for other areas of funds flowing through the system such as harvesting bribes.

Impact Explanation: Medium, because this will lead to inaccurate accounting and has implications for other areas of fund flows.

Likelihood Explanation: Low, because it is unlikely that governor will call the incorrect function.

Recommendation: Consider disallowing tokens with an amount greater than 0 in `$.protocolFeeAmounts[_token]` from being transferred through `Infrared.recoverERC20()`.

Infrared: Fixed in PR 303.

Spearbit: Reviewed that PR 303 resolves the issue as recommended.

5.4.9 Governance-enforced toggling between forced and voluntary withdrawals may be inefficient

Severity: Low Risk

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

Summary: Governance-enforced toggling of `withdrawalsEnabled` flag to switch between forced and voluntary withdrawals may be inefficient once voluntary withdrawals are enabled.

Finding Description: Given that only forced withdrawals are presently supported in the Berachain Consensus Layer, Infrared staking is also expected to currently support only that via `InfraredBERAWithdrawor.sweep()`.

However, there is initial support for voluntary withdrawals via `InfraredBERA.burn()` and in `InfraredBERAWithdrawor` (which has been highlighted in the issue "Unsupported functionality can be removed from staking withdrawal contracts to reduce attack surface)". The current design uses a `withdrawalsEnabled` flag to switch between forced and voluntary withdrawals. This flag value is set via `setWithdrawalsEnabled()` which is `onlyGovernor` access controlled. The default flag value of 0/false only allows forced withdrawals for now. This will be set to true when voluntary withdrawals are enabled. Thereafter, if/when there is a forced withdrawal as observed offchain by the Keeper, the governance is expected to set `withdrawalsEnabled` flag to false to allow the Keeper to sweep the forced withdrawn validator balance, and then set it back to true again to allow voluntary withdrawals (voluntary withdrawals are expected to be the norm and forced withdrawals the exception).

This governance-based toggling may lead to inefficiency: (1) Force-withdrawn BERA will sit idle until sweep is allowed by governance or (2) Voluntary withdrawals will be stalled while Keeper does the sweep and governance sets the flag to true again. The extent of inefficiency will depend on the governance process of voting on proposals and any enforced timelocks for proposal execution.

Impact Explanation: Low, because this only delays the forced/voluntary withdrawals resulting in slower redeployment of forced withdrawal balances or slower claims of voluntary withdrawals much after the burning of corresponding iBERA shares. Slower redeployment may lead to potential loss of any validator rewards in that window and slower claims may lead to potential loss of profits from trading the unclaimed BERA especially in volatile markets.

Likelihood Explanation: Low, because this initial support for voluntary withdrawals and the sweep functionality are both expected to be upgraded and refactored to better support the toggling. Also, forced withdrawals are expected to happen rarely, at least initially when Infrared validators are expected to dominate in the validator set both in number and deployed capital.

Recommendation: Consider a more efficient design where both forced/voluntary withdrawals are supported without toggling and without governance involvement. This depends on how Berachain BeaconKit implements/exposes verification of validator withdrawal reason (forced/voluntary) and corresponding balances, so that the forced/voluntary distinction can be made for required accounting.

Infrared: Acknowledged.

Spearbit: Acknowledged. Given that voluntary withdrawals are currently unsupported and that only `InfraredBERAWithdrawor.sweep()` is in scope now, this will have to be considered in future when support is added/scoped.

5.4.10 Deployer retaining privileged roles is risky

Severity: Low Risk

Context: InfraredDeployer.s.sol#L135, RED.sol#L25

Summary: Infrared deployer, which deploys `RED` and `IBGT` tokens, retaining privileged `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE` and `PAUSER_ROLE` roles is an unnecessary operational security risk.

Finding Description: `InfraredDeployer` script deploys the various Infrared contracts including `RED` and `IBGT` tokens. These tokens derive from `ERC20PresetMinterPauser`, which grants privileged `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE` and `PAUSER_ROLE` roles to `_msgSender()` i.e. `InfraredDeployer` script.

However, these privileged roles are meant to be granted only to `Infrared`. Allowing the `InfraredDeployer` script to also retain these privileged roles is an unnecessary operational security risk. There are many documented protocol exploits resulting from leaked deployer private keys.

Impact Explanation: High, because leaked deployer private key can allow, for example, infinite minting of `RED` and `IBGT` tokens.

Likelihood Explanation: Very low, assuming that deployer private key has the highest industry-standard operational security measures.

Recommendation: Consider self-revoking privileged roles for deployer after granting them to Infrared in their constructors.

Infrared: Fixed in PR 283.

Spearbit: InfraredDeployer.s.sol#L108-L110 appears to grant `MINTER_ROLE` to `data._gov` (governance) while it should really be `address(infrared)` to allow minting RED token in `harvestVault()`, correct?

```
red = new RED(
    address(ibgt), address(infrared), data._gov, data._gov, data._gov
);
```

This should be similar to what's done with iBGT, where `data._gov` has `DEFAULT_ADMIN_ROLE` and `PAUSER_ROLE`, but Infrared has `MINTER_ROLE`:

```
ibgt = new InfraredBGT(
    address(_bgt), data._gov, address(infrared), data._gov
);
```

Note: Infrared is also explicitly given `MINTER_ROLE` in the RED constructor via `_grantRole(MINTER_ROLE, infrared)` but I suppose that can be avoided by passing in Infrared instead of `data._gov` in the deployer for the `_minter` parameter.

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

Severity: Low Risk

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

Summary: Assigning both Keeper and Governance roles to Infrared `_admin` is risky because any compromise of Keeper can critically impact Governance actions.

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

Likelihood Explanation: 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.

Infrared: Fixed in PR 283.

Spearbit: Reviewed that PR 283 fixes the issue as recommended by separating the roles of Governance and Keeper into two addresses `_gov` and `_keeper` and granting them `_grantRole(DEFAULT_ADMIN_ROLE, _gov)` `_grantRole(GOVERNANCE_ROLE, _gov)` `_grantRole(KEEPER_ROLE, _keeper)` roles appropriately.

5.4.12 Non-whitelisted tokens cannot be recovered from Infrared vaults

Severity: Low Risk

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

Summary: Non-whitelisted tokens cannot be recovered from Infrared vaults.

Finding Description: Infrared `recoverERC20FromVault()` is meant for Governance to recover any ERC20 tokens stuck in the vaults. Ideally, this should be only allowed to recover non-whitelisted tokens from the vault because whitelisted tokens are already considered in the normal functioning of the protocol.

However, `VaultManagerLib.recoverERC20FromVault()` implements a flipped check to revert with `RewardTokenNotWhitelisted` while recovering any non-whitelisted token.

Impact Explanation: Low, because this only applies to non-whitelisted tokens stuck in the protocol either due to accidental transfers or if Governance fails to recover a whitelisted token before blacklisting it.

Likelihood Explanation: Low, because this only applies for accidental transfers or Governance misses.

Recommendation: Consider flipping the check to only allow recovery of non-whitelisted tokens. If not, document the requirement and usage scenarios.

Infrared: Fixed in PR 377.

Spearbit: This PR removes the `!isWhitelisted($, _token)` check, which allows recovery of any whitelisted token. Don't you want to disallow recovery of whitelisted tokens? Because, otherwise a malicious governance proposal can drain all such tokens.

5.4.13 Missing `whenInitialized` for `setVaultRegistrationPauseStatus()` allows it to be called before initialization

Severity: Low Risk

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

Summary: Missing `whenInitialized` modifier for `setVaultRegistrationPauseStatus()` allows it to be called before Infrared initialization.

Finding Description: All externally callable Infrared functions, including those with `onlyGovernor` modifier, have the `whenInitialized` modifier to prevent them from being accidentally called before the proxy implementation is initialized.

However, `setVaultRegistrationPauseStatus()` is missing such a `whenInitialized` modifier.

Impact Explanation: Low, because the rest of the protocol has not been initialized, which prevents any unexpected behavior.

Likelihood Explanation: Very Low, because the current deployer script calls `infrared.initialize()` atomically with Infrared deployment. Also, given the `onlyGovernor` modifier, it is very unlikely that Governance will call `setVaultRegistrationPauseStatus()` before initialization.

Recommendation: Consider adding `whenInitialized` modifier for `setVaultRegistrationPauseStatus()` to be consistent with the rest of the defensive checks.

Infrared: Acknowledged. `whenInitialized` has actually been removed now from all function in Infrared to reduce contract size and because it is a superfluous check given contracts will not work at all without initialization and is written into deployment scripts. See commit a273f3a0.

Spearbit: Acknowledged.

5.4.14 Staking token accidentally sent to an InfraredVault cannot be recovered

Severity: Low Risk

Context: MultiRewards.sol#L323-L331

Summary: The logic in `MultiRewards.sol` allows Infrared governance to recover any stray tokens (that are non-whitelisted) accidentally sent to an `InfraredVault`, but the `stakingToken` itself cannot be recovered from a vault.

Finding Description: The `_recoverERC20()` function in `MultiRewards.sol` is used when the governance tries to recover stray tokens residing in a vault. It disallows sweeping out any of the whitelisted reward tokens because those are being used in the normal functioning of the protocol, and it is meant to allow retrieving any other random tokens.

But it does not allow to retrieve any `stakingToken` balance sitting in the vault contract. Usually, the vault contract is not supposed to hold any `stakingToken` balance outside of the stake/ withdraw calls which immediately process the funds received.

So any `stakingToken` balance vaults have is supposed to be from accidental transfers only. It often happens that users trying to stake a token accidentally just "transfer" that token directly to the staking contract, which means vaults can have stray `stakingToken` balances.

Impact Explanation: Low, because its just stray token balance accidentally transferred to the vault.

Likelihood Explanation: Low, because this will only include users' mistake and is not significant to the functioning of the protocol.

Recommendation: Consider allowing the `stakingToken` to be recovered from the vaults, which can be later used to reimburse users or to aid protocol treasury as these are just donated funds.

Infrared: Fixed in PR 377.

Spearbit: Fixed. Any stray donations of stakingToken can now be recovered from MultiRewards (i.e. `InfraredVault`).

5.4.15 Potential underflow in `confirmed()` due to griefing donations may cause temporary DoS

Severity: Low Risk

Context: InfraredBERA.sol#L129

Summary: Potential underflow in `confirmed()` due to griefing donations may cause temporary DoS of protocol flows such as the withdrawor queue calls.

Finding Description: `confirmed()` calculates the confirmed deposits as:

```
return deposits - pending();
```

If `InfraredBERADepositor` receives a donation equal to or exceeding the `deposits` value, `pending()` can return a value greater than `deposits`. This will cause an underflow in the subtraction, reverting the transaction and effectively halting all operations relying on the `confirmed` value, such as the withdrawor queue calls (out-of-scope for this review).

While such a situation is unlikely unless the protocol has a low total deposit volume or high rebalancing activity, it remains a vulnerability that can be exploited under certain circumstances.

Proof of Concept:

```
function testQueueDonationUnderflow() public {
    uint256 fee = InfraredBERAConstants.MINIMUM_WITHDRAW_FEE + 1;
    uint256 amount = 1 ether;
    address receiver = alice;
    uint256 confirmed = ibera.confirmed();
    assertTrue(amount <= confirmed);

    vm.deal(address(ibera), 2 * fee);
    uint256 nonce = withdrawor.nonceRequest();

    vm.deal(address(depositor), 201 ether); // DONATION

    vm.prank(address(ibera));
    withdrawor.queue{value: fee}(receiver, amount);
}
```

Output:

```
            [822] iBERADepositor::fallback() [staticcall]
              [438] InfraredBERADepositor::reserves() [delegatecall]
                ← [Return] 20100000000000000000 [2.01e20]
              ← [Return] 20100000000000000000 [2.01e20]
            ← [Revert] panic: arithmetic underflow or overflow (0x11)
          ← [Revert] panic: arithmetic underflow or overflow (0x11)
        ← [Revert] panic: arithmetic underflow or overflow (0x11)
    ← [Revert] panic: arithmetic underflow or overflow (0x11)
 ← [Revert] panic: arithmetic underflow or overflow (0x11)

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 10.51ms (1.07ms CPU time)

Ran 1 test suite in 149.40ms (10.51ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in tests/unit/staking/InfraredBERAWithdrawor.t.sol:InfraredBERAWithdraworTest
[FAIL: panic: arithmetic underflow or overflow (0x11)] testQueueDonationUnderflow() (gas: 70662)
```

Impact Explanation: Medium, because if an underflow occurs in the `confirmed` function, it can cause the function to revert, halting critical protocol operations like withdrawor queue calls. This could temporarily disrupt the withdrawal process, affecting user experience and the reliability of the protocol. Furthermore, in extreme cases, this issue might necessitate manual intervention or contract redeployment, leading to reputational damage and operational overhead.

Likelihood Explanation: Low, because the scenario requires a griefing donation equal to or exceeding the `deposits` value, which is highly unlikely in normal operation. This situation could occur only if the protocol has very small total deposits, frequent large withdrawals, or excessive rebalancing. Additionally, this requires an actor intentionally sending a significant amount of BERA to the contract, making it a rare edge case.

Recommendation: Consider modifying `confirmed()` to prevent underflows when `pending()` exceeds `deposits`. This can be achieved by ensuring the function returns `0` in such cases:

```
function confirmed() external view returns (uint256) {
    uint256 pendingValue = pending();
    return deposits > pendingValue ? deposits - pendingValue : 0;
}
```

This approach ensures that the function will not revert due to underflow and maintains the protocol's operability even in extreme edge cases involving excessive donations.

Infrared: Fixed in PR 334.

Spearbit: Reviewed that PR 334 fixes the issue as recommended.

5.4.16 Inefficient fee handling during `InfraredBERADepositor` execute may lead to delayed deposits

Severity: Low Risk

Context: InfraredBERADepositor.sol#L165-L168

Description: The current implementation applies the full fee from a slip during its first use, regardless of whether the entire slip amount is consumed. This may lead to issues for keepers or users who process partial slip amounts in subsequent calls, as they would not receive any reimbursement for the fees they front. Furthermore, malicious actors could exploit this behavior by executing the minimal allowed slip amount, effectively draining the accumulated fees without fully processing the slip.

Impact Explanation: Medium, because improper handling of fees creates an imbalance in incentives for keepers or users, potentially leading to fewer participants willing to process deposits. This could delay protocol operations and harm user experience. Additionally, malicious actors exploiting the fee mechanism could deplete reserves, negatively affecting the protocol's functionality.

Likelihood Explanation: Low, because the issue depends on the frequency of partial slip executions and the protocol's reliance on external actors for deposit processing. While a highly technical malicious actor is needed to exploit the fee mechanism, the absence of an equitable fee distribution may deter honest participants from contributing.

Recommendation: Implement proportional fee deduction based on the amount of the slip actually used in the final iteration. This ensures that fees are fairly distributed among keepers processing the slip. This ensures that fees are equitably distributed and minimizes the risk of exploitation by malicious actors.

Infrared: Acknowledged. We think that the likelihood of a user being able to exploit is small with keeper having a 7 day head start to execute and only batch amounts of > 10k bera can be processed

Spearbit: Acknowledged.

5.4.17 Unpause functionality controlled by the same role as pause may be risky

Severity: Low Risk

Context: ERC20PresetMinterPauser.sol#L98-L104

Summary: The `unpause()` function in the `ERC20PresetMinterPauser` contract is controlled by the same `PAUSER_ROLE` as `pause()`, which may be risky.

Finding Description: The `unpause()` function in the `ERC20PresetMinterPauser` contract is controlled by the same `PAUSER_ROLE` as `pause()`. This means any account granted the ability to pause token transfers can also unpause them. In many governance or security-sensitive scenarios, unpausing is considered a higher-privilege operation because it restores the ability to transfer tokens, potentially impacting the system's security or governance intentions.

Impact Explanation: Medium, because allowing unpause with the same role can expose the system to misuse or unauthorized activity if the `PAUSER_ROLE` is granted to less-trusted individuals or entities. This is particularly concerning in systems with multiple pausers, as any pauser could reverse the intention of pausing.

Likelihood Explanation: Low, because this depends on how the protocol manages the assignment of the `PAUSER_ROLE`. In systems with careful role assignment and a small set of trusted administrators, the risk is reduced. However, in larger or decentralized setups, the risk increases as more accounts could have the `PAUSER_ROLE`.

Recommendation: Consider separating the roles for `pause` and `unpause` functionalities. A higher-privileged role, such as `OWNER_ROLE` or `ADMIN_ROLE`, should control the unpause functionality. This approach ensures stricter control over resuming token transfers, especially in scenarios where pausing might be initiated by less-trusted actors.

Infrared: Fixed in a273f3a0.

Spearbit: Reviewed that a273f3a0 grants `PAUSER_ROLE` to governance, which will control both pausing and unpausing.

5.5 Gas Optimization

5.5.1 Gas Savings

Severity: Gas Optimization

Context: VaultManagerLib.sol#L90, MultiRewards.sol#L297, InfraredBERAWithdrawor.sol#L159, DelegationLogicLibrary.sol#L52-L58, Voter.sol#L290

Description:

1. DelegationLogicLibrary.checkpointDelegator incurs extra storage writes when two calls in same block as `cp` is written to storage and then moved after the `_isCheckpointInNewBlock` check. To save the duplicate, the check could be used to determine which index is the appropriate one to write to.
2. Voter._vote should just be = since above is already read and enforced to be 0 at line 284 `if (votes[_tokenId][_stakingToken] != 0) revert NonZeroVotes();`.

3. InfraredBERAWithdrawor.execute, and `Depositor` both read a storage nonce in a while loop. It would be more efficient to use a memory variable and write it to storage once the loop is complete.
4. MultiRewards._notifyRewards reading `rewardData[_rewardsToken].rewardResidual;` after writing it to storage means the compiler will not cache it. Would be more efficient to cache the value then use it for both the write and then the subtraction on L297.
5. VaultManagerLib.addReward performs the same argument validation as the downstream call to the `InfraredVault` does: `if (_rewardsDuration == 0) revert Errors.ZeroAmount();`
6. Reward.earned velodrome uses `Math.max` saving the cost of calculating epochStart 2x.
7. Reward._withdra if the new balanceOf is calculated and cached before writing to storage (`balanceOf[tokenId] = newChachedAmount;`) the cached value can be used instead of reading from storage after update. The same optimization opportunity exists in `_deposit`.

Recommendation: Consider making the above listed optimizations.

5.5.2 Cached `currentOperator` can be reused instead of a repeated `getOperator()` external call

Severity: Gas Optimization

Context: InfraredBERADepositor.sol#L191

Description: `InfraredBERADepositor.execute()` caches `IBeaconDeposit(DEPOSIT_CONTRACT).getOperator(pubkey)` in `currentOperator`. However, it later makes a second call to the same external function to get the current operator for `pubkey`.

Recommendation: Reuse cached `currentOperator` to save gas.

Infrared: Fixed in PR 375.

5.5.3 `shareholderFees` can be cached to avoid repeated storage reads

Severity: Gas Optimization

Context: InfraredBERAFeeReceivor.sol#L77-L78

Description: `InfraredBERAFeeReceivor.collect()` reads `shareholderFees` from storage for the sanity check in `shareholderFees == 0`. However, instead of caching this storage value, it reads it again immediately in `shf = shareholderFees`.

Recommendation: Cache `shareholderFees` in a local variable before the sanity check and use that to avoid repeated storage reads and save gas.

Infrared: Fixed in PR 318.

5.5.4 Redundant checks can be removed

Severity: Gas Optimization

Context: InfraredVault.sol#L124-L125, InfraredVault.sol#L141-L142, InfraredVault.sol#L154-L155, VaultManagerLib.sol#L155, VaultManagerLib.sol#L159, InfraredBERADepositor.sol#L132, InfraredBERAFeeReceivor.sol#L70

Description: Redundant checks across functions in `InfraredVault` and `VaultManagerLib` can be carefully removed to save gas.

Recommendation:

1. Zero checks for `_rewardsToken` and `_rewardsDuration` in `InfraredVault.updateRewardsDuration()` are redundant given the similar checks in `VaultManagerLib.updateRewardsDurationForVault()`.
2. Zero checks for `_rewardsToken` and `_rewardsDuration` in `InfraredVault.addReward()` are redundant given the similar checks in `VaultManagerLib.addReward()`.

3. Zero checks for `_rewardToken` and `_reward` in `InfraredVault.notifyRewardAmount()` are redundant given the similar checks in `VaultManagerLib.addIncentives()`.
4. `_amount == 0` is redundant in `VaultManagerLib.recoverERC20FromVault()` given a similar check in `vault.recoverERC20()`.
5. `_to == address(0)` is redundant in `VaultManagerLib.recoverERC20FromVault()` given a similar check in `vault.recoverERC20()`.
6. `!IInfraredBERA(InfraredBERA).validator(pubkey)` is redundant in `InfraredBERADepositor.execute()` given a similar check earlier in the function.
7. `amount > 0` is redundant in `InfraredBERAFeeReceivor.sweep()` given the `amount < min` check earlier in the same function.

Infrared: Fixed in PR 322.

5.5.5 Early Fee Validation in burn Function

Severity: Gas Optimization

Context: InfraredBERA.sol#L205

Description: The line:

```
uint256 fee = msg.value;
```

could benefit from an early validation to ensure that the fee is greater than or equal to `MINIMUM_WITHDRAW_FEE`. Currently, this validation is deferred to the `queue` call, which may waste computation resources before reverting. Adding this check earlier would fail the transaction immediately, saving unnecessary processing and improving efficiency.

Recommendation: Introduce an early validation in the `burn` function to check if `msg.value` is greater than or equal to `MINIMUM_WITHDRAW_FEE`. This prevents redundant operations and improves user feedback by failing transactions as soon as possible in cases of insufficient fees.

5.5.6 Redundant Ownership Check

Severity: Gas Optimization

Context: Voter.sol#L468

Description: In the `claimBribes` function, a check is performed to ensure the caller is the approved owner of the `_tokenId` using the `IVotingEscrow.isApprovedOrOwner` function. However, this same check is redundantly repeated inside each call to `getReward` of the `IReward` contract when a `VotingReward` implementation is used. This leads to unnecessary repeated validations, incurring additional gas costs for each iteration of the loop.

Recommendation: To eliminate redundant checks, the `getReward` function in the `VotingReward` implementation can be access-controlled to ensure it is only callable from the `Voter` contract. By implementing Access Control Logic (ACL) and restricting direct user calls, the `isApprovedOrOwner` check inside `getReward` can be safely removed. This would make `Voter` the single point of entry for authorization checks, thereby maintaining security while optimizing gas usage. Also, this means that all calls to the `getReward` function should first check for the `isApprovedOrOwner` as that will no longer be inside and a external check is required.

5.5.7 `InfraredBERADepositor.execute()` loop needs to be optimized to avoid OOG

Severity: Gas Optimization

Context: InfraredBERAConstants.sol#L6

Description: Stakers are allowed to deposit `MINIMUM_DEPOSIT` amounts, which is currently configured to be `0.1 ether`. if stakers only deposit this minimum amount then we will need, in the worst case scenario, 320+ stakers

to queue up deposits before keeper can trigger `InfraredBERADepositor.execute()` with an amount > `MIN_DEPOSIT_AMOUNT_IN_GWEI == 32 ether` to meet Berachain deposit contract's minimum deposit amount.

Recommendation: `InfraredBERADepositor.execute()` loop needs to be gas optimized to avoid OOG in worst case scenarios for final values of `MINIMUM_DEPOSIT` and `MIN_DEPOSIT_AMOUNT_IN_GWEI`.

Infrared: Fixed in PR 340.

5.5.8 Reward calculations should use cached values in MultiRewards.sol

Severity: Gas Optimization

Context: MultiRewards.sol#L81-L82, MultiRewards.sol#L151-L163

Description: There are several instances in the rewards calculation where cached values can be used to save gas. Examples are:

- In `updateReward()`, `lastTimeRewardApplicable()` is called twice here : one for storing `lastUpdateTime` and one inside `rewardPerToken()` logic ⇒ `rewardPerToken()` can be modified to return the `lastUpdateTime` also and use this value to store `lastUpdateTime`.
- All the calculation in `updateReward()` for a particular token can be skipped when `rewardData[token].lastUpdateTime == block.timestamp` ⇒ will save gas on a lot of calculations.
- At MultiRewards.sol # L85 ⇒ can cache the value of rewardPerTokenStored obtained at Line 81 above, to be used here again. Instead of reading from the state, we already have the latest value so we can use it here.
- In earned() ⇒ `earned()` is called from `updateReward()` where `rewardPerToken` calculation is already done. all flows whether stake/withdraw/claim etc... follow the same flow ⇒ first update `rewardPerToken` and then calculate earned. So earned here can safely use the stored `rewardPerToken` value ie. `rewardData[rewardToken].rewardPerTokenStored`.

Recommendation: Consider applying recommended changes.

5.5.9 Unnecessary fee calculations on RED rewards can be avoided

Severity: Gas Optimization

Context: RewardsLib.sol#L190-L195

Description: `RewardsLib.harvestVault()` performs fee calculations on RED rewards with a call to `_chargedFeesOnRewards()` using zero values for `_feeTotal` and `_feeProtocol`, which returns 0 for `_amtVoter` and `_amtProtocol`. This is followed by a call to `_distributeFeesOnRewards()`. Thees calculations are unnecessary. `ProtocolFees` event emit is the only side-effect.

Recommendation: Consider removing these fee calculations on RED rewards unless non-zero values are planned to be used for fees.

5.5.10 Event emit can use parameter `_rewardsDuration` instead of state variable

Severity: Gas Optimization

Context: MultiRewards.sol#L356-L358

Description: `MultiRewards._setRewardsDuration()` emits an event `RewardsDurationUpdated` where the updated `rewardData[_rewardsToken].rewardsDuration` is used.

Recommendation: Use parameter `_rewardsDuration` instead of the state variable updated with its value to save gas from SLOAD.

5.6 Informational

5.6.1 Missing access control on `InfraredBERA.sweep()` allows donations

Severity: Informational

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

Description: `InfraredBERA.sweep()` is meant to be only callable by `InfraredBERAFeeReceivor.sweep()` so that accumulated coinbase priority fees and MEV are deposited to validators for auto-compounding the principal staked.

However, there is no access control on `InfraredBERA.sweep()` to restrict caller to `InfraredBERAFeeReceivor.sweep()`. This allows anyone to donate towards validator deposits but does not appear to have any other unexpected side-effects.

Recommendation: Consider restricting `InfraredBERA.sweep()` caller to `InfraredBERAFeeReceivor` as a defensive measure.

Infrared: Fixed in PR 327.

Spearbit: Reviewed that PR 327 fixes the issue as recommended.

5.6.2 Unsupported functionality can be removed from staking withdrawal contracts to reduce attack surface

Severity: Informational

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

Description: "*BeaconKit does not currently support voluntary withdrawals, nor slashing or inactivity leaks.*" from the `validator-lifecycle` section of Berachain's docs. Given this current lack of support at the Berachain Consensus Layer, the staking withdrawal related functionality in `InfraredBERAWithdrawor`, `InfraredBERAClaimor` and `InfraredBERA.burn()` is potentially incomplete, not expected to be used and therefore is enforced with `withdrawalsEnabled == false`. Only `InfraredBERAWithdrawor.sweep()` is supported for forced withdrawal validators to re-stake principal.

As such, only `InfraredBERAWithdrawor.sweep()` is considered as in-scope for this review. Rest of the logic in `InfraredBERAWithdrawor`, `InfraredBERAClaimor` and `InfraredBERA.burn()` are out-of-scope.

Recommendation: Consider removing the currently unsupported and potentially incomplete functionality from staking withdrawal contracts to reduce attack surface. They can up added in future contract upgrades when voluntary withdrawals are supported.

Infrared: Fixed in PR 302.

Spearbit: Reviewed that PR 302 fixes the issue as recommended by replacing `InfraredBERAWithdrawor` with `InfraredBERAWithdraworLite`, which retains only the `sweep()` function. Note: `InfraredBERAWithdrawor` non-sweep related logic/flows, `InfraredBERAClaimor` and their new upgrade script `UpgradeInfraredBERAWithdrawor` are out-of-scope for this review.

5.6.3 Withdrawal sweep check can be stricter

Severity: Informational

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

Description: Keeper is expected to call `sweep(uint256 amount, bytes calldata pubkey)` for a validator with `pubkey` and effective balance of `amount` when it detects offchain that such a validator has been subjected to forced withdrawal from the Berachain Consensus Layer. In `sweep()`, there is a sanity check where it reverts if `amount > address(this).balance`. However, the amount being swept can never be greater than `stakes(pubkey)` because otherwise `IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount))` will revert.

Recommendation: Consider changing `amount > address(this).balance` to `amount > stakes(pubkey)` to enforce a stricter check.

Infrared: Fixed in PR 324.

Spearbit: Reviewed that PR 324 avoids the issue by refactoring the logic as noted:

1. Streamlined the stake withdrawal process by removing the need for an external amount parameter.
2. Introduced a mechanism to track validators that have force exited, enhancing validator management.
3. Added functionality to check if a validator has exited.

5.6.4 Incorrect address emitted in `Sweep` event

Severity: Informational

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

Description: The `Sweep(address indexed receiver, uint256 amount)` event emitted in `InfraredBERAWithdrawor.sweep()` is expected to log the `receiver` address for the amount being swept. However, `emit Sweep(InfraredBERA, amount)` uses `InfraredBERA` as the `receiver` instead of `IInfraredBERA(InfraredBERA).depositor()`, which is where the swept funds are sent to be queued.

Recommendation: Consider replacing `InfraredBERA` with `IInfraredBERA(InfraredBERA).depositor()` in the `Sweep` event emit.

Infrared: Fixed in PR 298.

Spearbit: Reviewed that PR 298 fixes the issue as recommended.

5.6.5 Inconsistent convention for internal function naming with _ prefix

Severity: Informational

Context: MultiRewards.sol#L200, MultiRewards.sol#L222, MultiRewards.sol#L245, IReward.sol#L267, IReward.sol#L275

Description: Each of the 3 `on` functions are missing the _ prefix which is used for internal functions throughout the codebase.

Conversly, the prefix is used for two of the `IReward` public functions.

Recommendation: Prefix the internal functions with _ and omit from public functions.

Infrared: Agree with this comment. These contracts come from the velodrome fork, which we are reluctant to change too much as it has already been audited and battle tested in production but we can add these amendments to the list. See IReward.sol#L76-L80 on commit 9e5a5748.

Spearbit: Acknowledged.

5.6.6 Use typed argument for fee type on `fees` getter function

Severity: Informational

Context: Infrared.sol#L700

Description: It would be more consistent to keep the argument typed `ConfigTypes.FeeType _t`.

5.6.7 `Infrared.update*` functions perform the update if the new value matches the old value

Severity: Informational

Context: Infrared.sol#L272

Description: The 3 `Infrared.update*` functions updateWhiteListedRewardTokens, updateRewardsDuration, updateRewardsDurationForVault still perform the update if the new value matches the old value.

Recommendation: Add an idempotent check to revert on repeat calls ensuring the events are not emitted multiple times.

5.6.8 Outdated comment from forked codebase

Severity: Informational

Context: BalanceLogicLibrary.sol#L88

Description: The comment `/// @dev Adheres to the ERC20` balanceOf `interface for Aragon compatibility` refers to the `balanceOf` function and is true for Curve but not for Solidly and derivatives.

Recommendation: Correct the comment and leave the implement as is.

Infrared: Fixed in commit 111fc420.

Spearbit: Reviewed that commit 111fc420 fixes the issue by adding a clarifying comment: "*Although only true of curve, but not solidly and its forks*".

5.6.9 Withdrawals always hitting the Consensus Layer is sub-optimal

Severity: Informational

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

Description: Voluntary withdrawals (which is out-of-scope), as currently implemented, do not tap into `pending()` funds on the protocol, which includes `depositor.reserves()` and `withdrawor.rebalancing()` amounts. This effectively forces withdrawals to always hit the Consensus Layer every time even when they can be satisfied with pending funds available within the protocol.

Recommendation: Consider redesigning this logic so that voluntary withdrawals can tap into protocol-available pending funds when possible for a better UX.

5.6.10 Incorrect/Stale/Incomplete comments are misleading and reduce code comprehension

Severity: Informational

Context: RewardsLib.sol#L491, VaultManagerLib.sol#L25, VaultManagerLib.sol#L34, VaultManagerLib.sol#L128, MultiRewards.sol#L212, InfraredBERADepositor.sol#L83, InfraredBERADepositor.sol#L189, InfraredBERAFeeReceivor.sol#L15, InfraredBERA.sol#L76, Voter.sol#L248, VotingEscrow.sol#L75

Description: There are some incorrect/stale/incomplete comments in the codebase which are misleading and should be fixed/removed to improve code comprehension.

Recommendation:

1. `// TODO: check correct`: TODO's indicate potentially incomplete logic and should be checked/removed after due consideration.
2. `* @param _token Address of token (VELO) used to create a veNFT`: `VELO` (from the forked Velodrome codebase) should be replaced by `RED`
3. `// burn minimum amount to mitigate inflation attack with shares`: This should say mint, not burn.
4. `/// @notice Fee receivor receives coinbase priority fees + MEV credited to contract on EL upon block validation`: Should add "*+ validator bribe share*".
5. `// @dev can be called by withdrawor when rebalancing`: Should add "*and sweeping*".
6. `// hook withdraw then transfer staking token out, in case hook needs to bring in collateral` : this is incorrect because the flow always brings in the exact amount of staked token from berachain rewardVault, should be changed to "*hook withdraw then transfer staking token out*".
7. `If _isPoke is true, deposit in fees reward vault in addition to marking tokenId as voted`: this is incorrect and should say "*Skip processing for tokens with killed bribe vaults*".

8. `Reverts if the caller is not the collector`: this is irrelevant and could be removed.
9. `Registers a new vault for a specific asset with specified reward tokens`: Wrong comment because there are no "*specified reward tokens*" while creating a new vault. It just gets deployed with iBGT as a rewardToken by default.
10. `Updates the rewards duration for vaults`: Misleading comment because it actually updates the global rewardsDuration value used for configuration when deploying "*new*" vaults
11. `// uint256 _redAmt = Math.mulDiv(_bgtAmt, $.redMintRate, RATE_UNIT);`: This is outdated because the current implementation foregoes `Math.mulDiv` in favor of `uint256 redAmt = bgtAmt * mintRate / RATE_UNIT;`.

Infrared: Fixed in PR 381.

Spearbit: Reviewed that PR 381 fixes the issue as recommended.

5.6.11 Unused code reduces code comprehension

Severity: Informational

Context: MultiRewards.sol#L9, DataTypes.sol#L36, Errors.sol#L4

Description: There are unused declarations and other constructs in the codebase which reduce code comprehension and should be removed. Example include:

1. Except `NATIVE_ASSET`, all other datatypes in `src/utils/DataTypes.sol` are unused.
2. Many errors in `src/utils/Errors.sol` including `ZeroBytes` and `OverFlow` are unused.
3. `import {Ownable}` in `src/core/MultiRewards.sol` is unused.

Recommendation: Removed unused code to improve code comprehension.

Infrared: Fixed in PR 369.

Spearbit: PR 369 fixes (1) related to `src/utils/DataTypes.sol` but not (2) and (3).

5.6.12 Missing event emit in `updateRedMintRate()`

Severity: Informational

Context: Infrared.sol#L404

Description: Infrared `updateRedMintRate()` is a Governance-controlled function to update the RED minting rate, which determines how many RED tokens are minted per IBGT. However, this privileged function is missing an event emit for transparency and monitoring.

Recommendation: Consider adding an event emit in `updateRedMintRate()`.

Infrared: Fixed in PR 383.

Spearbit: Reviewed that PR 383 fixes the issue as recommended by adding a `UpdatedRedMintRate` event emit.

5.6.13 Direct hashing of pubkey Instead of using `getValidatorId()` is inconsistent

Severity: Informational

Context: ValidatorManagerLib.sol#L26, ValidatorManagerLib.sol#L66, ValidatorManagerLib.sol#L84, ValidatorManagerLib.sol#L119, ValidatorManagerLib.sol#L165, ValidatorManagerLib.sol#L182

Description: The contract directly derives a validator ID by hashing the `pubkey` using `keccak256` in multiple locations instead of using `_getValidatorId()` provided in the library. While functionally correct, this approach introduces potential maintainability issues. If the logic for deriving the validator ID changes in the future (e.g., switching from `keccak256` to another hashing mechanism), these changes would need to be updated across all occurrences instead of a single centralized implementation.

Recommendation: Consider replacing all instances of `keccak256(pubkey)` with `_getValidatorId(pubkey)`. This ensures that any future changes to the ID derivation logic are centralized within the `_getValidatorId` function, improving maintainability and reducing the potential for bugs in upgrades.

Infrared: Acknowledged.

Spearbit: Acknowledged.

5.6.14 Missing sanity check on `_pubkeys` in `cancelBoosts()` and `cancelDropBoosts()` is inconsistent

Severity: Informational

Context: ValidatorManagerLib.sol#L144-L157, ValidatorManagerLib.sol#L190-L203

Description: `ValidatorManagerLib` functions which accept validator `_pubkeys` apply a sanity check to verify that they are indeed `validatorIds`. However, `cancelBoosts()` and `cancelDropBoosts()` are missing such a check. While this does not cause unexpected behavior (because the underlying `IBerachainBGT` functions operate on `msg.sender`, which is Infrared and therefore affect only Infrared validators), it is inconsistent with other similar functions.

Recommendation: Consider adding a sanity check on `_pubkeys` in `cancelBoosts()` and `cancelDropBoosts()`.

Infrared: Fixed in PR 317.

Spearbit: Reviewed that PR 317 fixes the issue as recommended.

5.6.15 Validation checks across `remove()`, `purge()`, and `claim()` functions are inconsistent

Severity: Informational

Context: InfraredDistributor.sol#L67, InfraredDistributor.sol#L81, InfraredDistributor.sol#L113

Description: In the `InfraredDistributor` contract:

1. `remove()`: The `amountsCumulative == 0` check is redundant because `amountsCumulative` is initialized to `1` during contract deployment and cannot be decremented. This check does not provide meaningful validation.
2. `purge()`: The `s.amountCumulativeLast == 0` check is unnecessary, as `amountCumulativeLast` is initialized during `add` and cannot be zero. Instead, the function should check if `s.amountCumulativeFinal == 0` to ensure only removed validators can be purged. The existing `s.amountCumulativeLast != s.amountCumulativeFinal` check is sufficient to prevent purging validators with unclaimed rewards.
3. `claim()`: The `s.amountCumulativeLast == 0` check is also unnecessary because `amountCumulativeLast` is initialized during `add`. Instead, the function should validate that `s.amountCumulativeLast` is not equal to `fin`, ensuring the validator has unclaimed rewards and is not in a purgeable state, thereby preventing griefing and event spamming.

Recommendation: Remove the unnecessary checks in `remove()`, `purge()`, and `claim()`, and replace them with validations that align with the intended behavior of these functions. Specifically:

- In `remove()`, avoid redundant checks and ensure `amountCumulativeFinal` is not already set.
- In `purge()`, validate `amountCumulativeFinal` to confirm the validator has been removed explicitly.
- In `claim()`, ensure there are unclaimed rewards by comparing `amountCumulativeLast` and `fin` to prevent unnecessary operations and potential misuse.

This ensures consistency, reduces redundant logic, and safeguards the intended functionality of the contract.

5.6.16 Misleading names of `validators()` and `snapshots()` do not reflect their functionality

Severity: Informational

Context: InfraredDistributor.sol#L131, InfraredDistributor.sol#L141

Description: The `validators` function name is misleading because it suggests that it returns a list of all validators, while in reality, it retrieves only the validator address associated with a specific `pubkey`. Similarly, the `snapshots` function name implies that it provides all snapshots for validators but instead returns the snapshot data for a single validator identified by a specific `pubkey`. These naming inconsistencies can cause confusion for developers and users, leading to potential misunderstandings about the intended functionality of these functions.

Recommendation: The `validators` function could be renamed to `validator` to reflect its purpose of fetching the information of a single validator. Likewise, the `snapshots` function could be renamed to `snapshot` to indicate that it retrieves the snapshot data for a specific validator. These adjustments would improve clarity and align the function names with their actual behavior, enhancing overall code readability and usability.

Infrared: Fixed in PR 385.

Spearbit: Reviewed that PR 385 fixes the issue as recommended by adding a `get` prefix to their names.

5.6.17 Dead code in `MultiRewards.sol` can be removed

Severity: Informational

Context: MultiRewards.sol#L140-L142

Description: The `rewardPerToken()` calculation has a branch of logic for when `totalSupply` of `infraredVault == 0`.

This branch will be unreachable because each `InfraredVault` will always have a `totalSupply >= 1` as on deployment, 1 wei of stake is added for the Infrared contract. Hence this is dead code.

Recommendation: Consider removing this branch of logic.

5.6.18 Missing Natspec params reduce code comprehension

Severity: Informational

Context: Infrared.sol#L496, InfraredBERAFeeReceivor.sol#L26-L28, Voter.sol#L255

Description: There are some functions whose Natspec is incomplete, which reduces code comprehension.

Recommendation:

1. `Voter._vote()` is missing `param` for the newly added parameter `_isPoke`.
2. `InfraredBERAFeeReceivor.initialize()` is missing `param` for `ibera` and `_infrared`.
3. `Infrared.harvestOperatorRewards()` is missing from `interface IInfrared`.

5.6.19 `VotingReward.notifyRewardAmount()` is missing `nonReentrant` modifier

Severity: Informational

Context: Reward.sol#L299

Description: `Reward.notifyRewardAmount` uses a `nonReentrant` modifier. It is then overridden in `VotingReward` where the modifier is removed. Finally it is overridden again in `BribeVotingReward` where the modifier is added back again.

Recommendation: If all three contracts are intended to be used, consider adding the modifier to `VotingReward` as well. Alternatively, the base contract security checks can be exposed by using the pattern of calling `super.notifyRewardAmount` when extending the Reward contract.

Infrared: Fixed in PR 387.

Spearbit: Reviewed that PR 387 resolves the issue by adding the modifier to `VotingReward` as well.

5.6.20 Misleading comment about initial reward tokens

Severity: Informational

Context: InfraredVault.sol#L56

Description: The comment in the constructor states:

```
// add initial reward tokens requiring at least IBGT and IRED
```

However, the code only adds `IBGT` as an initial reward token and does not add `IRED`. This discrepancy between the comment and the actual implementation can confuse developers or auditors, leading to misunderstandings about the initial state of the reward tokens in the vault.

This should reflect what the code is doing. If only `IBGT` is intended to be added as an initial reward token, the comment should clearly state that. If `IRED` is also intended to be added, the code should include the logic to add `IRED` as a reward token during initialization.

Recommendation: Clarify the comment or code to reflect the current behavior of the code/comment accurately.

Infrared: Fixed in PR 299.

Spearbit: Reviewed that PR 299 removes the comment.

5.6.21 Misalignment in reward calculation in `getAllRewardsForUser()` may cause unexpected behavior

Severity: Informational

Context: InfraredVault.sol#L181-L207

Description: The `getAllRewardsForUser` function calculates rewards for a user based on the last `lastUpdateTime` recorded on-chain rather than the current block's timestamp. Unlike the `updateReward` function, it does not update `lastUpdateTime`, which impacts the `rewardPerToken` calculation. This results in the `earned` values reflecting outdated rewards, leading to under-reported rewards for users when `getAllRewardsForUser` is called.

This behavior can cause confusion among users and stakeholders, as the function does not accurately reflect the rewards the user is entitled to at the moment of the call.

Recommendation: To ensure `getAllRewardsForUser` returns accurate and up-to-date reward amounts, the function should mimic the behavior of `updateReward` by factoring in the latest block's timestamp for reward calculations. This can be achieved by temporarily updating the `lastUpdateTime` and using the current block's timestamp to calculate the `rewardPerToken` and earned values. However, this must be done without persisting state changes, as `getAllRewardsForUser` is a view function.

An alternative approach is to document the function's behavior explicitly, clarifying that it reflects rewards based on the last recorded `lastUpdateTime` and not the most recent block rewards. This can help manage user expectations while leaving the implementation unchanged.

Infrared: Fixed in PR 372.

Spearbit: Reviewed that PR 372 fixes the issue by documenting appropriately as recommended.

5.6.22 Missing zero-address validation for `ibgt`

Severity: Informational

Context: Infrared.sol#L183

Description: In the `Infrared` contract constructor, the line `ibgt = IInfraredBGT(_ibgt);` assigns the `_ibgt` address to the `ibgt` immutable variable without checking if `_ibgt` is a valid or non-zero address. If `_ibgt` is mistakenly set to the zero address during contract deployment, all vaults registered through the `registerVault` function will add a reward token with the zero address, leading to significant issues in reward distribution and contract behavior.

This could result in incorrect or failing reward logic, potential exploits, or unusable vaults. Since `_ibgt` is immutable, there is no way to correct the mistake post-deployment.

Recommendation: Add a validation check to ensure `_ibgt` is not zero-address during contract deployment. If `_ibgt` is invalid, the constructor should revert. This ensures that the protocol cannot be deployed with an invalid `ibgt` address, preventing cascading errors across the vault system.

Infrared: Fixed in commit a273f3a0.

Spearbit: Reviewed that commit a273f3a0 fixes the issue as recommended.

5.6.23 Redundant logic in updating `rewardsDuration`

Severity: Informational

Context: Infrared.sol#L201-L203

Description: The lines:

```
if (_rewardsDuration == 0) revert Errors.ZeroAmount();

_vaultStorage().rewardsDuration = _rewardsDuration;
```

are redundant and could be replaced with a call to `_vaultStorage().updateRewardsDuration(_rewardsDuration)`, leveraging the existing `updateRewardsDuration()` function in the `VaultManagerLib` library. This would reduce duplication and improve maintainability by centralizing the logic for updating the `rewardsDuration`.

Recommendation: Refactor the code to call `_vaultStorage().updateRewardsDuration(_rewardsDuration)` directly. This ensures consistency, avoids redundancy, and makes use of the modular design provided by the library.

5.6.24 `chargedFeesOnRewards` getter not performing input validation is inconsistent with the corresponding setter

Severity: Informational

Context: Infrared.sol#L411-L423

Description: `Infrared.chargedFeesOnRewards()` getter performs no input validation and will respond to inputs that are not possible in practice given the `RewardsLib.updateFee()` validation that enforces the `_fee` to be less than or equal to `FEE_UNIT`.

Recommendation: Consider adding validation mirroring the `updateFee` setter: `if (_fee > FEE_UNIT) revert Errors.InvalidFee();`.

Infrared: Fixed in PR 394.

Spearbit: Reviewed that PR 394 fixes the issue as recommended.

5.6.25 `RewardsLib.updateRedMintRate` performs no validation on the `_iredMintRate` argument

Severity: Informational

Context: RewardsLib.sol#L493

Description: `RewardsLib.updateRedMintRate` performs no validation on the `_iredMintRate` argument, which could allow setting it to unreasonable values.

Recommendation: Consider validating for an upper limit to stop unreasonable values from being set.

Infrared: Acknowledged. Team has elected to include a check in governance script, which is a work in progress.

Spearbit: Acknowledged.

5.6.26 `previewMint` and `previewBurn` return incorrect fee values for reverting scenarios

Severity: Informational

Context: InfraredBERA.sol#L294, InfraredBERA.sol#L320

Description: `InfraredBERA`'s `previewMint` and `previewBurn` preview the amount of InfraredBERA shares that would be minted/burned for a given BERA amount. They also return the fee that would be charged for the operations. For reverting/failure scenarios, they return a zero amount but mistakenly return a value of `fee == InfraredBERAConstants.MINIMUM_DEPOSIT_FEE`.

Recommendation: Consider returning `fee == 0` for reverting/failure scenarios.

Infrared: Fixed in PR 301.

Spearbit: There may be two other places where this needs to be fixed:

1. InfraredBERA.sol#L360.
2. InfraredBERA.sol#L377.

But given these two are in `previewBurn()` related to `burn` which is OOS, we will consider PR 301 as fixing this issue for now.

5.6.27 `InfraredBERADeployer` script is redundant

Severity: Informational

Context: InfraredBERADeployer.s.sol#L16

Description: `InfraredBERADeployer` deploys all the staking contracts related to `InfraredBERA`. However, `InfraredDeployer` also deploys those contracts along with the core Infrared ones, which makes `InfraredBERADeployer` script redundant.

Recommendation: Consider removing the redundant `InfraredBERADeployer` script.

Infrared: Fixed in PR 302.

Spearbit: Reviewed that PR 302 fixes the issue as recommended.

5.6.28 Missing event emits in `Infrared.initialize()`

Severity: Informational

Context: Infrared.sol#L210-L227

Description: `Infrared.initialize()` calls `updateWhitelistedRewardTokens()` and `registerVault()` via `_vaultStorage()` and missed emitting related events as done in their respective setters.

Recommendation: Consider emitting `WhiteListedRewardTokensUpdated` and `NewVault` events for transparency and monitoring.

Infrared: Fixed in PR 296 and PR 297.

Spearbit: Reviewed that PR 296 and PR 297 fix the issue as recommended.

5.6.29 The zero amount check in `execute()` is redundant and suboptimal

Severity: Informational

Context: InfraredBERADepositor.sol#L154

Description: The `s.amount == 0` check in `execute()` appears redundant because the `amount` is already validated in the `queue` function at line 91. However, rather than simply removing this check, it is better to replace it with a more meaningful validation to ensure the `nonce` is valid.

The current logic indirectly checks for invalid `nonce` scenarios where `nonce == nonceSlip`. A clearer and more meaningful check would verify:

```
if (nonce >= nonceSlip) revert Errors.InvalidNonce();
```

This approach directly validates whether the `nonce` is within the valid range, improving readability and providing more specific error messaging.

Recommendation: Replace the current redundant check with a validation for `nonce` being less than `nonceSlip` as illustrated earlier. This ensures that the function explicitly checks for invalid nonce scenarios and provides clearer feedback in case of a failure.

5.6.30 Misleading variable name `feeShareholders` does not reflect its functionality

Severity: Informational

Context: InfraredBERAFeeReceivor.sol#L55

Description: The variable `feeShareholders` in the `InfraredBERAFeeReceivor` contract is used to calculate a percentage fee as a fraction of the total balance, defined as `1/feeShareholders`. This approach is efficient and minimizes storage cost, but the variable name can be misleading as it does not explicitly reflect its purpose as a divisor for calculating fractional fees.

For instance:

```
fees = amount / uint256(feeShareholders);
```

The name `feeShareholders` might imply a relationship with shareholders or their total count, while its actual purpose is to act as a divisor for fee calculation.

Recommendation: Rename the variable to better reflect its purpose. Suggested names include:

- `feeDivisorShareholders`.
- `feeDenominatorShareholders`.

This change will improve the readability and clarity of the contract, making its logic easier to understand for future developers and auditors.

Infrared: Fixed in PR 273.

Spearbit: Verified that in PR 273 the variable is renamed as suggested.

5.6.31 Redundant zero check can be removed

Severity: Informational

Context: InfraredBERAFeeReceivor.sol#L81

Description: The conditional statement on line 81 of the `InfraredBERAFeeReceivor` contract contains a redundant `shf == 0` check. This is unnecessary because the earlier condition on line 77 already handles it vua `shareholderFees == 0`. If `shareholderFees` (and consequently `shf`) is 0, the function would have already returned by the time execution reaches line 81.

Furthermore, the condition `shf == 0` is inherently covered by `shf < min`, as 0 is within the range `[0, min)`. Thus, the `shf == 0` check is entirely redundant.

Recommendation: Consider simplifying the conditional statement by removing the redundant `shf == 0` check. This change ensures that the code remains concise and avoids unnecessary evaluations.

Infrared: Fixed in commit a6724604.

Spearbit: Reviewed that commit a6724604 fixes the issue as recommended.

5.6.32 Invalid hardcoded EIP-7002 Precompile address

Severity: Informational

Context: InfraredBERAWithdrawor.sol#L22

Description: The address `0x00A3ca265EBcb825B45F985A16CEFB49958cE017` is hardcoded as the precompile address for the EIP-7002 Withdraw Precompile, which is intended to handle Ethereum withdrawals. However, according to the EIP-7002 specification, the standard address for this precompile on Ethereum is expected to be `0x0c15F14308530b7CDB8460094BbB9cC28b9AaaAA`. The discrepancy between the hardcoded address and the standard address specified in the EIP may cause interoperability and functionality issues.

Recommendation: Verify the correct precompile address based on the network and EIP-7002 implementation being targeted. Update the contract to use the corresponding address based on the deployment parameters if it needs to be adjusted and document how the address is obtained from.

Infrared: Fixed in PR 302.

Spearbit: Reviewed that PR 302 fixes the issue with a new initializer function `initializeV2()`, which will allow setting the actual `WITHDRAW_PRECOMPILE` when the network is up.

5.6.33 `Errors.NoRewardsVault` should be used instead of `Errors.VaultNotSupported` for consistency

Severity: Informational

Context: VaultManagerLib.sol#L65-L66, VaultManagerLib.sol#L173-L174, VaultManagerLib.sol#L188-L189

Description: If `vaultRegistry[asset] == address(0)` then `Errors.NoRewardsVault` is used to indicate that there is no associated rewards vault in the registry for that asset. However, there are two places where `Errors.VaultNotSupported` in used instead.

Recommendation: Consider using `Errors.NoRewardsVault` instead of `Errors.VaultNotSupported` for consistency.

5.6.34 Modifying variable names to better reflect actual functionality will improve code comprehension

Severity: Informational

Context: Infrared.sol#L248, Infrared.sol#L309, VaultManagerLib.sol#L25, VaultManagerLib.sol#L115-L118, MultiRewards.sol#L55, MultiRewards.sol#L277, InfraredBERADepositor.sol#L185, InfraredBERAFeeReceivor.sol#L75, InfraredBERAWithdrawor.sol#L34-L37

Description: At several places in the codebase, names of variables/ events / errors can be modified to better reflect the actual functionality.

There are several instances of this :

- `Infrared.sol :: addReward()` ⇒ Could be named `addRewardsToken()` because its adding a new reward token and add reward implies notifying some amount of rewards.
- `MultiRewards.sol` :: `rewards` mapping ⇒ Could rename this to `userRewardPerTokenUnclaimed` as all accrued rewards for a user are stored here until they are claimed.
- `MultiRewards.sol` ::`_addReward()` :: event `RewardStored` ⇒ something like "REWARD TOKEN ADDED" would be a better event here.
- `Infrared.sol :: pauseVault()` ⇒ Consider renaming to `togglePause()`.
- `InfraredBERADepositor.sol` ⇒ Consider renaming `credentials` to `withdrawal_credentials`.
- `InfraredBERAFeeReceivor.sol` :: `collect()` ⇒ Can be renamed to `collectFees()`.
- `InfraredBERAWithdrawor.sol` :: `struct Request {}` ⇒ Consider renaming `amountSubmit` as `amountPendingSubmit` and `amountProcess` as `amountPendingProcess` to better reflect usage.

- `VaultManagerLib.sol` :: `addIncentives()` :: error `RewardTokenNotWhitelisted` ⇒ In case the token has not been added to a vault, the correct error here would be `RewardTokenNotSupported()`.

Recommendation: Consider applying recommended changes.

5.6.35 There is no way to remove a reward token from a vault even if it is removed from the whitelist

Severity: Informational

Context: Infrared.sol#L259-L265

Summary: A reward token that is removed from Infrared's whitelist cannot be removed from an infrared vault's configuration once it is added.

Finding Description: A reward token that is once whitelisted and configured (added) to an `InfraredVault` will exist there in the `rewardTokens` mapping forever even after the `rewardToken` is removed from the `whitelistedTokens` list at the Infrared contract level.

Since `addIncentives()` is a public function, anyone can call this and it will notify rewards for that specific `rewardToken` (and involve token transfer) that has now been removed from the whitelist at Infrared level but still exists in the list for an `InfraredVault` that was configured with it.

So this will always allow sending that `rewardToken` to the vault, which might have greater impact if the token was malicious (or becomes malicious and is not whitelisted).

This may lead to unintended consequences when these tokens are interacted with from the vault.

Recommendation: Consider disallowing `addIncentives()` call for a token that has been removed from the whitelist. Also consider adding this check to `VaultManagerLib.addIncentives()`.

5.6.36 Fee changes might be retroactively applied to unharvested rewards

Severity: Informational

Context: Infrared.sol#L367-L375

Description: Fees is charged when rewards are harvested. For example, `harvestVault()` calculates the amount of new BGT emissions received and mints iBGT and RED rewards (and takes a small portion of these as the protocol fees). All reward mechanisms have their own fee configurations.

There is a way for the governance to update this fee configuration via `Infrared.sol :: updateFee()`.

Since the associated code does not handle any unharvested rewards, if the fee gets updated when there are pending rewards to be harvested, then the next time `harvestVault()` (or related functions are called), the new fees will be applied to those rewards that were earned prior to the fee change.

This might lead to over/under charging of the applicable fees.

Recommendation: Document that a call to `updateFee()` should always be preceded by a call to harvest the rewards associated with that fee type, or make necessary code changes to first harvest rewards before updating fees.

5.6.37 Unused function parameter in `chargedFeesOnRewards()` can be removed

Severity: Informational

Context: RewardsLib.sol#L53-L54

Description: This function is used to calculate the relevant fees charged on a certain amount of rewards earned, and returns the portion of funds that have to be directed to `voter fee vault` as well as the protocol fee component.

The `RewardsStorage` is passed as a storage reference to this function, but because this is a pure function, it does not make use of this variable and directly does the calculations.

Recommendation: Remove this function parameter from the declaration as it is unnecessary.