Infrared contracts
Security Review

Cantina Managed review by:

Optimum, Lead Security Researcher
R0bert, Lead Security Researcher

February 18, 2025

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

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

From Jan 21st to Feb 1st the Cantina team conducted a review of infrared-contracts on commit hash 65de7c25. The team identified a total of **23** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	5	2	3
Low Risk	11	5	6
Gas Optimizations	0	0	0
Informational	6	3	3
Total	**23**	**11**	**12**

3 Findings

3.1 High Risk

3.1.1 Possible underflow in `harvestBase` caused by burning IBGT tokens

Severity: High Risk

Context: RewardsLib.sol#L118

Description: The `harvestBase` function calculates surplus BGT by subtracting `minted = iBGT.totalSupply()` from the `Infrared` contract's on-chain BGT balance. If a user burns some iBGT tokens, that minted value drops, creating what appears to be a larger surplus (`balance - minted`). However, if all actual BGT in the `Infrared` contract has already been "boosted", the `Infrared` contract will fail when it attempts to redeem that surplus BGT. The BGT contract's `checkUnboostedBalance` modifier ensures only unboosted BGT can be redeemed, so the attempt to redeem what it thinks is newly unlocked BGT actually reverts. The LSD effectively counts any burned iBGT as if it frees up new BGT to redeem, but the BGT remains boosted and thus unavailable, causing a revert during the `BGT.redeem` call.

At this point, any call to `harvestBase` would revert.

Recommendation: Consider making iBGT token not burnable or use a new state variable that is updated whenever iBGT tokens are minted to track the exact amount minted over time.

Infrared: Fixed in PR 501.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Users can "snipe" newly arrived bribes at an outdated fixed price

Severity: Medium Risk

Context: Infrared.sol#L481

Description: The `BribeCollector` contract employs a static `payoutAmount` mechanism, allowing any user to purchase all bribe tokens within the contract by paying a set amount of `payoutToken`. While governance can update this price by calling `setPayoutAmount`, there is no requirement that newly arrived bribes remain locked until the price is updated. If a significant quantity of bribe tokens accumulate before governance can adjust `payoutAmount`, an opportunistic user can claim them cheaply by calling `claimFees` with the previously valid, but now outdated, `payoutAmount`.

The fact that the `Infrared.harvestBribes` function is permissionless allows any user to pull bribes into the `BribeCollector` whenever they are available in the Infrared contract.

The moment large bribes enter the `BribeCollector`, they become immediately purchasable at whatever the old `payoutAmount` happens to be. If governance is even slightly delayed in noticing the large influx of bribes, an external caller may "snipe" the tokens, paying far less than their fair value.

Recommendation: Consider incorporating a time delay or timelock on newly arrived bribes in the `BribeCollector.claimFees` function, preventing them from being claimed for a set number of blocks or seconds after they are pulled from the Infrared contract. This would give governance or a keeper sufficient opportunity to observe changes in the contract's bribe balance and adjust `payoutAmount` accordingly before the tokens become available.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 `setIBGT` and `setRed` calls can be front-run

Severity: Medium Risk

Context: Infrared.sol#L403-L430

Description: The `Infrared` contract's `setIBGT` and `setRed` functions can each only be called once. These functions are permissionless and yet they permanently assign the `Infrared`'s token references. If these functions are not invoked atomically with the initial `Infrared.initialize` transaction, a malicious user could quickly call `setIBGT` or `setRed` ahead of the deployer, registering any random or malicious token addresses. That would lock the protocol into using those addresses. The current deployment script (`InfraredDeployer.s.sol`) calls `Infrared.initialize`, then makes separate calls to `setIBGT` and `setRed` in subsequent transactions, leaving a window of time for an attacker to front-run those calls.

Recommendation: Include the `setIBGT` and setRed logic `directly` in the same transaction as `Infrared.initialize`, for example, by having a single “`initAll`” function in `Infrared` that sets iBGT/red references upon initialization, or by combining them in a single script-based atomic approach (e.g., using a contract-based deployment aggregator). This ensures no gap exists where a malicious third party can call `setIBGT` or `setRed` with a hostile token. Moreover, ensure that the `Infrared` contract deployment and initialization calls are also executed atomically.

Infrared: Acknowledged and mostly prevented as `setIBGT` is called in deployment script `script/InfraredDeployer.s.sol`, which will be batched with other deployments txs for atomic execution using `multicall`. `setRed` has actually since been changed to be `onlyGovernor`.

Cantina Managed: Acknowledged.

3.2.3 Immediate share minting in `InfraredBERA.mint` dilutes yield when validator max capacity is reached

Severity: Medium Risk

Context: InfraredBERA.sol#L185

Description: The current logic in `InfraredBERA.mint` immediately mints new `iBERA` shares to a user as soon as they deposit BERA. However, if all the validators have reached their maximum capacity (`MAX_-EFFECTIVE_BALANCE`), the user's deposit is merely queued and never actually staked. That deposit sits in the `InfraredBERADepositor` queue, not generating more "validator rewards". Yet the user is already holding fresh iBERA shares and diluting the rewards for everyone else, because the total iBERA supply rose while no new stake was introduced.

The system gives newly minted iBERA for a deposit that fails to become actual staking collateral. As yield from real staked BERA continues to flow, more shares are now in circulation, reducing the share of yield per iBERA token. Other holders who truly have staked BERA suffer from this unbacked issuance. The deposit will remain in the queue until either capacity frees up, new validators are added or some forced re-routing occurs.

Recommendation: Add a validator capacity check in the `InfraredBERA.mint` function to ensure that the user's BERA deposit can actually be staked. Calculate the total remaining capacity across all validators and if the deposit would exceed this aggregate unused capacity, revert the transaction. For this calculation do consider also the total amounts present in both deposit and withdrawal queues.

Infrared: Acknowledged. In its current state we could have a maximum of around 250 validators with max balances of 10 million bera each, totalling 2.5 billion bera. Only after that would this edge case kick in. Likely if we reach half of this, we can arrange an upgrade to reject more mints.

Cantina Managed: Acknowledged.

3.2.4 `InfraredBERADepositor.execute` might allow deposits to forced exited validators

Severity: Medium Risk

Context: InfraredBERADepositor.sol#L109

Description: The `InfraredBERADepositor` contract is used by the `InfraredBERA` contract to facilitate deposits to Bera validators. The deposit process is a two-step process in which funds are queued first and then used by either the keeper or the original depositor (after a week) upon a call to `execute` that make the deposit to the Berachain deposit contract.

In Berachain there is no slashing mechanism but still validators can be forced exit and their stake will be sent to the `InfraredBERAWithdrawor` contract. The `sweep` function handles these funds by allowing the keeper to re-stake these to another validator. However, the `sweep` function can only be called when withdrawals are disabled (governance decision to set `InfraredBERA.withdrawalsEnabled` to false) which makes the scenario of depositing to an exited/withdrawn validator possible since the `InfraredBERADepositor` lacks a necessary check for that.

Recommendation: With the current implementation it is challenging to think about a mitigation that will completely solve this issue. However, for the purpose of this review we will focus on proposing a mitigation for `InfraredBERAWithdraworLite` (and not for `InfraredBERAWithdrawor`) which will make it easier. The proposed solution will be to add a check to `InfraredBERADepositor.execute` that ensures deposits are allowed only in case the balance of `InfraredBERAWithdraworLite` is less than `InfraredBERAConstants.INITIAL_DEPOSIT`. This will not be a perfect solution since attackers may decide to donate 10k BERA tokens to the withdrawer contract to cause a DoS of deposits. This attack be mitigated by adding another function that is only accessible by governance to withdraw the malicious deposit.

Infrared: Fixed in PR 473.

Cantina Managed: Fix verified.

3.2.5 Lack of access control

Severity: Medium Risk

Context: InfraredDistributor.sol#L89

Description:

1. The `InfraredDistributor` contract implements the function `purge` used to purge the validator from the registry completely:

```
function purge(bytes calldata pubkey) external {
    address validator = _validators[keccak256(pubkey)];
    if (validator == address(0)) revert Errors.ValidatorDoesNotExist();

    Snapshot memory s = _snapshots[keccak256(pubkey)];
    if (s.amountCumulativeLast != s.amountCumulativeFinal) {
        revert Errors.ClaimableRewardsExist();
    }

    delete _snapshots[keccak256(pubkey)];
    delete _validators[keccak256(pubkey)];

    emit Purged(pubkey, validator);
}
```

   However, this function is permissionless and therefore anyone can call it. The only condition is that `s.amountCumulativeLast` is equal to `s.amountCumulativeFinal` which will be met after the validator's operator calls the `claim` function. Thus, any user will be able to purge a validator from the registry right after the validator had claimed its rewards.

2. The `harvestBase` function calculates surplus BGT by subtracting minted = `iBGT.totalSupply()` from the `Infrared` contract's on-chain BGT balance. If a user burns some iBGT tokens, that minted value drops, creating what appears to be a larger surplus (`balance - minted`). However, if all actual BGT in the `Infrared` contract has already been "boosted", the `Infrared` contract will fail when it attempts to redeem that surplus BGT. The BGT contract's `checkUnboostedBalance` modifier ensures only unboosted BGT can be redeemed, so the attempt to redeem what it thinks is newly unlocked BGT actually reverts. The LSD effectively counts any burned iBGT as if it frees up new BGT to redeem, but the BGT remains boosted and thus unavailable, causing a revert during the `BGT.redeem` call.

   At this point, any call to `harvestBase` would revert.

Recommendations:

1. Consider adding some sort of access control to this function.
2. Consider making iBGT token not burnable or use a new state variable that is updated whenever iBGT tokens are minted to track the exact amount minted over time.

Infrared: Fixed in commit 39428dc9.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 Race condition in `BribeCollector`'s payoutAmount

Severity: Low Risk

Context: BribeCollector.sol#L62

Description: The `BribeCollector` relies on a single storage variable `payoutAmount` to determine how many tokens a user must pay to claim all bribes. A user may observe `payoutAmount` on-chain (e.g., `1e18`) and grant a large token allowance to the contract for that amount, then call `claimFees` in the same block that governance calls `setPayoutAmount` to increase the `payoutAmount` to (e.g., `2e18`). If the governor's transaction is processed first, the `payoutAmount` is updated to `2e18` by the time the user's `claimFees` is executed, forcing them to pay the higher price despite them anticipating the lower 1e18 cost. Because the user has pre-approved a max allowance, there is no further prompt or revert preventing the newly required higher payment.

Recommendation: Consider having the user supply their expected `payoutAmount` as a parameter to `claimFees` and revert if it does not match the contract's current `payoutAmount`. This ensures the user's transaction can only succeed at the exact price they observed on-chain.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 Immediate zeroing of user rewards if ERC20 transfer fails

Severity: Low Risk

Context: MultiRewards.sol#L236

Description: Inside `MultiRewards.getRewardForUser`, the contract sets `rewards[_user][_rewardsToken]` to zero before confirming that the actual ERC20 transfer call succeeded. If the ERC20 token is paused, blacklisted, or otherwise fails the transfer call, the user still ends up with their `rewards[_user][_rewardsToken]` reduced to zero, meaning they cannot subsequently reclaim those reward tokens once the issue is resolved. This results in permanent loss of unclaimed rewards for the user. The current logic discards or "burns" the user's accrued balance by zeroing it out prematurely, even when the actual token transfer never takes place.

Recommendation: Only clear `rewards[_user][_rewardsToken]` after confirming that the `transfer` call fully succeeds. This way, if the token contract is temporarily paused, the reward remains unclaimed and can be attempted again later once normal transfers are possible.

Infrared: Fixed in PR 401.

Cantina Managed: Fix verified.

3.3.3 `VaultManagerLib.addIncentives`: fee-on-transfer and rebase tokens can not be used as incentive tokens

Severity: Low Risk

Context: VaultManagerLib.sol#L117

Description: `VaultManagerLib.addIncentives` uses the "pull-pattern" of `safeTransferFrom` pulling `_amount` from the `msg.sender` and later transferring this amount to the chosen vault. In case the provided `_rewardsToken` is a fee-on-transfer or a rebase token, the call to `vault.notifyRewardAmount` will revert for insufficient balance.

Recommendation: Consider either documenting that these type of exotic tokens are not meant to be supported in your docs or alternatively use the diff of token balance calculated by querying the contract balance before and after the call to `safeTransferFrom` instead of `_amount` for the call to `vault.notifyRewardAmount`.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.3.4 `InfraredBERAWithdrawor.sweep` might be called with the wrong `pubkey` which may delay deposits

Severity: Low Risk

Context: InfraredBERAWithdrawor.sol#L266

Description: `sweep` is used to re-deposit funds that were sent to the contract from forced-exit validators. The function is restricted to the keeper only, however, in the current implementation the received funds are not earmarked to the specific validator which may cause an accounting error in case the keeper is injecting the wrong `pubkey`. Consider the following scenario:

Two active validators A, B. At some point A is forced exit, and the staked funds are transferred to the `InfraredBERAWithdrawor` contract. At this point, assuming `sweep` is called with the pubkey of B, the call will succeed. The issue will arise later when the keeper will try to call the `execute` function with the pubkey of B which will fail since B will be considered exited in the `_exited` mapping. In order to solve the issue the keeper will have to call `execute` with the pubkey of A instead.

Recommendation: As suggested in the "*TODO*" comment in the code, consider verifying the withdrawal against beacon roots.

Infrared: Acknowledged. We are planning to implement this in a future release.

Cantina Managed: Acknowledged.

3.3.5 Potential improvements for `InfraredBERAWithdrawor`

Severity: Low Risk

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

Description: The `InfraredBERAWithdrawor` contract is still in "WIP" state. We had noticed some points that should be considered during future development of this contract:

1. `InfraredBERAWithdrawor.process()`: The function is processing withdrawal requests sequentially, which means that in case a user is willing to call `process` to receive its funds he might need to call this function many times until he will be able to fulfill his request. We can not rely on the user to get into implementation details that's why we propose to either allow a "random-access" processing of funds or make sure to operate a script that calls `process` as a service for the users.
2. The withdrawal process is comprised of a few steps. The initiation of the process starts with calling `InfraredBERA.burn()` which will revert in case `withdrawalsEnabled` is false. That is not the case for the rest of the steps in the withdrawal process such as `execute` and `process`.
3. The contract allows the keeper to rebalance, i.e. withdraw funds from a validator and re-stake it. This feature is using the same functions used for user withdrawals, however, both withdrawal and re-deposit fees will be taken from the keeper, we think it is not needed and should be removed.
4. Withdrawals are based only on funds already staked in validators and do not utilize deposits that have not been staked yet and are still pending in the `InfraredBERADepositor` contract.
5. `sweep` makes sure that `amount > address(this).balance` which does not take into account the fees accumulated that can not be used to fulfil withdrawals. Therefore `amount > reserves()` should be used instead.

Infrared: Acknowledged, however some of the improvements were already implemented like the `amount > reserves()` in the `sweep` function.

Cantina Managed: Acknowledged.

3.3.6 `InfraredBERA`: `feeDivisorShareholders` should be above a certain threshold

Severity: Low Risk

Context: InfraredBERA.sol#L247

Description: As we can see in the following code snippet, the value of `feeDivisorShareholders` does not have a minimum value.

```
function setFeeDivisorShareholders(uint16 to) external onlyGovernor {
    emit SetFeeShareholders(feeDivisorShareholders, to);
    feeDivisorShareholders = to;
}
```

`feeDivisorShareholders` is later used in the following code snippet, in theory it can be set to 1 (100%) which will allocate all the fees to the `shareholderFees` and is not in favor of stakers.

```
function distribution()
    public
    view
    returns (uint256 amount, uint256 fees)
{
    amount = (address(this).balance - shareholderFees);
    uint16 feeShareholders =
        IInfraredBERA(InfraredBERA).feeDivisorShareholders();

    // take protocol fees
    if (feeShareholders > 0) {
        fees = amount / uint256(feeShareholders);
        amount -= fees;
    }
}
```

Recommendation: Consider enforcing a minimum value for `feeDivisorShareholders`.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.3.7 `VaultManagerLib`: reward tokens removed from whitelist can still be used as incentives

Severity: Low Risk

Context: VaultManagerLib.sol#L79

Description: The `MultiRewards` contract allows adding reward tokens that can be transferred to the contract to incentivize staking. However, the contract does not implement a mechanism to remove rewards, especially in case these were removed from the whitelist maintained in `VaultManagerLib.updateWhitelistedRewardTokens()`. To put it more simple, in case a token was whitelisted and added to a certain vault, but then later was removed from the whitelist, it is still possible to add units from these tokens as incentives to vaults as we can see in:

```
function addIncentives(
    VaultStorage storage $,
    address _stakingToken,
    address _rewardsToken,
    uint256 _amount
) external {
    if (address($.vaultRegistry[_stakingToken]) == address(0)) {
        revert Errors.NoRewardsVault();
    }

    IInfraredVault vault = $.vaultRegistry[_stakingToken];

    (, uint256 _vaultRewardsDuration,,,,,) = vault.rewardData(_rewardsToken);
    if (_vaultRewardsDuration == 0) {
        revert Errors.RewardTokenNotWhitelisted();
    }

    ERC20(_rewardsToken).safeTransferFrom(
        msg.sender, address(this), _amount
    );
    ERC20(_rewardsToken).safeApprove(address(vault), _amount);

    vault.notifyRewardAmount(_rewardsToken, _amount);
}
```

Recommendation: Consider either adding functionality to remove reward tokens from vaults, which will need to recover the funds currently held by the contract. In case this solution is an overkill for you, consider restricting `addIncentives` only for whitelisted tokens.

Infrared: Fixed in PR 532.

Cantina Managed: Fix verified.

3.3.8 `InfraredBERA.setFeeDivisorShareholders`: share holders can claim larger portion of the fee retroactively

Severity: Low Risk

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

Description: `setFeeDivisorShareholders` allows the governor of the contract to set the portion of fees that will be allocated to share holders. This percentage can change at any given time which means the change may apply retroactively in case `compound()` was not called for a while.

Recommendation: Consider adding a call to `compound()` inside `setFeeDivisorShareholders` like other functions of this contract.

Infrared: Fixed in commit ea7a361d.

Cantina Managed: Fix verified.

3.3.9 Innacurate validator forced exit check in `InfraredBERADepositor` contract

Severity: Low Risk

Context: PR 513.

Description: The `InfraredBERADepositor` contract assumes that there is any forced exits on the withdrawor contract if the balance of the withdrawor is more than `INITIAL_DEPOSIT`:

```
// cache the withdrawor address since we will be using it multiple times.
address withdrawor = IInfraredBERA(InfraredBERA).withdrawor();

// Check if there is any forced exits on the withdrawor contract.
// @notice if the balance of the withdrawor is more than INITIAL_DEPOSIT, we can assume that there is an
↪  unprocessed forced exit and
// we should sweep it before we can deposit the BERA. This stops the protocol from staking into exited
↪  validators.
if (withdrawor.balance >= InfraredBERAConstants.INITIAL_DEPOSIT) {
    revert Errors.HandleForceExitsBeforeDeposits();
}
```

However, by checking directly the `withdrawor.balance`, the withdrawal fees (1e18 per withdrawal queued) are also counted.

Recommendation: Update the `InfraredBERADepositor` check as shown below:

```
- if (withdrawor.balance >= InfraredBERAConstants.INITIAL_DEPOSIT) {
+ if ((withdrawor.balance -  IInfraredBERAWithdrawor(withdrawor).fees()) >=
↪  InfraredBERAConstants.INITIAL_DEPOSIT) {
```

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.3.10 Native asset address holder is still whitelisted in the Infrared contract

Severity: Low Risk

Context: PR 463.

Description: In the reviewed PR 463 the following code was removed from the `RewardsLib.harvestBribes` function:

```
if (_token == DataTypes.NATIVE_ASSET) {
    IWBERA(wbera).deposit{value: address(this).balance}();
    _token = wbera;
}
```

However, in the `Infrared.harvestBribes` function `DataTypes.NATIVE_ASSET` is still whitelisted.

Recommendation: Consider updating the `Infrared.harvestBribes` function as shown below:

```
for (uint256 i; i < len; ++i) {
    if (
        whitelistedRewardTokens(_tokens[i])
    ) {
        whitelisted[i] = true;
    }
}
```

Infrared: Fixed in PR 492.

Cantina Managed: Fix verified.

3.3.11 `MultiRewards.getRewardForUser`: potential denial of service by reward tokens causing out of gas

Severity: Low Risk

Context: MultiRewards.sol#L237

Description: `MultiRewards.getRewardForUser` distributes all rewards allocated for a user iterating through the storage array `rewardTokens`. The function continues to handling the next token in case of a failed call to `ERC20.transfer` which is the correct way to handle multiple transfers but does not take into account tokens that might implement a gas heavy `transfer` function causing the entire transaction to revert and therefore a denial of service that will result in user rewards being stuck in the contract.

Recommendation: The mitigation strategy for this issue should consist 3 things:

1. Limit the gas that's forwarded to the call, run gas tests for some well-known implementations of `ERC.transfer`, as far as we concern 200,000 should be sufficient.
2. Implement a function to allow users to claim rewards only for a single token, this can be used as an emergency function. Keep in mind that claimed rewards should be cleaned from `rewards[_user][_rewardsToken]`.
3. Conduct a vetting process for external tokens. For the complete list of tasks for this process refer to vetting-process-for-external-tokens.md.

Infrared: Fixed in commit a0d44e64.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Missing values inside events

Severity: Informational

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

Description:

1. InfraredBERADepositor.sol#L201: The `Execute` event should also include the value of `fee`.
2. InfraredBERAWithdrawor.sol#L211: The `Execute` event should also include the value of `excess`.
3. InfraredBERAWithdraworLite.sol#L151: The `receiver` of the `Sweep` event should be `IInfraredBERA(InfraredBERA).depositor()`.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.4.2 Gas optimizations and redundant code

Severity: Informational

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

Description/Recommendation:

1. InfraredBERADepositor.sol#L62: The `KEEPER_ROLE` is not used in this contract, and instead, the keeper of `InfraredBERA` is used.
2. InfraredBERAClaimor.sol#L32: The `KEEPER_ROLE` is not used in this contract.
3. `InfraredUpgradeable` constructor is calling `_disableInitializers()` which is redundant since `Upgradeable` empty constructor is called before which already calls `_disableInitializers()`.
4. `Infrared`: Many storage variables can be declared immutable instead.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.

3.4.3 `InfraredBERADepositor.queue()`: Slips based deposit mechanism should be revised

Severity: Informational

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

Description: The function implements a deposit queue which resembles the queue used in `InfraredBERAWithdrawor`. After reviewing the entire repository, we think that this approach introduces redundant complexity that increases surface area. One particular scenario that we tested internally is the scenario of stuffing the deposit queue with minimum deposits which will increase the gas costs for the keeper, it is important to mention that a potential attacker will have to pay a cost for deposit in the current code which will decrease the probability of such an attack.

Recommendation: As outlined in complete-guide-to-securing-smart-contracts/code-conservatism.md the goal in smart contract security is to keep the code as concise as possible while still meeting all functional requirements. In this context, the slips mechanism should be eliminated, and deposits should instead be aggregated into a single shared variable. If the deposit fee is intended to deter potential DoS attacks, it can be entirely removed under the proposed solution, simplifying the system further.

Infrared: Fixed in commit ea7a361d.

Cantina Managed: Fix verified.

3.4.4 Handling storage of base contracts in case of upgrades

Severity: Informational

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

Description: The current repository consists of many upgradeable contracts that inherit base contracts. The default linear storage in solidity is constructed so that the compiler allocates the first slots to base contracts and then to the derived contracts. In a future scenario where the team will need to add variables to base contracts the team will not be able to add linear based storage variables since new variables will override the derived contracts variables. In order to handle this you can either add the well known `__gap` variable as mentioned in storage-gaps or to leave the code as is and add eip-7201 storage variables to base contracts instead. For more information refer to complete-guide-to-securing-smart-contracts/unstructured-storage.md.

Infrared: Fixed in commit a0d44e64.

Cantina Managed: Fix verified.

3.4.5 Use of balance checks instead of BGT Staker return value

Severity: Informational

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

Description: The `RewardsLib.harvestBoostRewards` function checks the token balance before and after the `_bgtStaker.getReward()` call to determine the exact reward amount received. However, this exact value is given directly as a return of the `_bgtStaker.getReward()` call.

Recommendation: Consider using the return value of the `_bgtStaker.getReward()` call to determine the exact amount of rewards received.

Infrared: Fixed in PR 493.

Cantina Managed: Fix verified.

3.4.6 `replaceValidator` calls can be front-run

Severity: Informational

Context: Infrared.sol#L571

Description: The `Infrared` contract implements the `replaceValidator` function. This function will remove first a validator from the `InfraredDistributor` contract before adding a new one:

```
function remove(bytes calldata pubkey) external onlyInfrared {
    address validator = _validators[keccak256(pubkey)];
    if (validator == address(0)) revert Errors.ValidatorDoesNotExist();

    uint256 _amountsCumulative = amountsCumulative;
    if (_amountsCumulative == 0) revert Errors.ZeroAmount();

    Snapshot storage s = _snapshots[keccak256(pubkey)];
    // Add check to prevent re-removal of already removed validators
    if (s.amountCumulativeFinal != 0) {
        revert Errors.ValidatorAlreadyRemoved();
    }

    s.amountCumulativeFinal = _amountsCumulative;

    emit Removed(pubkey, validator, _amountsCumulative);
}
```

The following check, prevents removing any validator if any reward was sent to the `InfraredDistributor` contract:

```
uint256 _amountsCumulative = amountsCumulative;
if (_amountsCumulative == 0) revert Errors.ZeroAmount();
```

As the `notifyRewardAmount` function is permissionless any user could sent a very small dust amount of tokens to the contract to DoS replacing or removing any validator.

Recommendation: Consider adding access control to the `InfraredDistributor.notifyRewardAmount` function.

Infrared: Acknowledged.

Cantina Managed: Acknowledged.