CANTINA

Infrared contracts

Competition

February 4, 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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 15th to Jan 28th Cantina hosted a competition based on infrared-contracts. The participants identified a total of **236** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 18
- Medium Risk: 42
- Low Risk: 110
- Gas Optimizations: 0
- Informational: 66

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Attackers can freeze user rewards

Severity: High Risk

Context: MultiRewards.sol#L236

Summary: The `MultiRewards` contract, which is inherited by the `InfraredVault`, contains the `getRewardForUser` function that allows any user to get rewards for a specific user. However, the amount of reward is set to zero even if the `transfer` call fails, which can be abused by attackers.

Finding Description: The rewards are transferred to the user via the low-level `call` function (see here), and if the call were to fail for any reason, the function will just continue to the next reward. This is done so that reverting `transfer` calls don't block transfers of other tokens which could be transferred successfully.

The problem lies in the accounting of the contract. The function first sets the `rewards` of the reward token for the user to zero (as we can see here) and calls the `transfer` function after that. As we can observe in the code, no action is taken on the call fail, and the function proceeds to process the next reward token.

However, the reward amount was set to zero while no tokens were transferred. This can happen for multiple reasons; the token might, for example, have a pausable function and be paused at the moment of claim. Since the call is permissionless, malicious users could leverage this to claim rewards of other users while the specific reward tokens are untransferable, effectively freezing their rewards.

While this would mean a lower likelihood due to the token constraints, attackers can also leverage the 63/64 gas rule and call the `getRewardForUser` function with insufficient gas. In the case of one reward token, this would lead to a revert of the `transfer` call, even if it was meant to pass, while the reward of the user was set to zero.

Impact Explanation: High. Long-term freeze of unclaimed yield.

Likelihood Explanation: Medium. The attacker might not profit in any way from the actions; however, the cost of such an attack is negligible.

Recommendation: Consider setting the rewards to zero only on successful transfer. While this breaks the CEI pattern, the function is non-reentrant, so this change should not impact the protocol.

```diff
    function getRewardForUser(address _user)
        public
        nonReentrant
        updateReward(_user)
    {
        onReward();
        uint256 len = rewardTokens.length;
        for (uint256 i; i < len; i++) {
            address _rewardsToken = rewardTokens[i];
            uint256 reward = rewards[_user][_rewardsToken];
            if (reward > 0) {
-               rewards[_user][_rewardsToken] = 0;
                (bool success, bytes memory data) = _rewardsToken.call(
                    abi.encodeWithSelector(
                        ERC20.transfer.selector, _user, reward
                    )
                );
                if (success && (data.length == 0 || abi.decode(data, (bool)))) {
+                   rewards[_user][_rewardsToken] = 0;
                    emit RewardPaid(_user, _rewardsToken, reward);
                } else {
                    continue;
                }
            }
        }
    }
```

3.1.2 Add/remove/replace validators can be DoSed becuase of a revert in `InfraredBERAFeeReceivor::collect`

Severity: High Risk

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

Description: When "governor" wants to add, remove, or replace validators, one of the following functions are called `Infrared::addValidators`, `Infrared::removeValidators`, and `Infrared::replaceValidator`. All of those functions harvest operator rewards before doing any validator operations, by calling:

```
harvestBase();
harvestOperatorRewards();
```

Looking into `harvestOperatorRewards`, it calls `harvestBoostRewards` function in the `Rewards` library, https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/core/libraries/RewardsLib.sol#L310-L329.

~~~
    function harvestOperatorRewards(
        RewardsStorage storage $,
        address ibera,
        address voter,
        address distributor
    ) external returns (uint256 _amt) {
1@>     IInfraredBERA(ibera).compound();
2@>     uint256 iBERAShares = IInfraredBERA(ibera).collect();

        if (iBERAShares == 0) return 0;

        // ...
    }
````.
It first calls `InfraredBERA::compound`, as seen in `1@`, which sweeps the rewards that are accumulated in the
↪  `InfraredBERAFeeReceivor`, this includes depositing some of that amount as a reinvestment in the pool,
↪  https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/staking/InfraredBERAFeeReceivor.sol#L78.
↪  The other part of that is called "shareholder fees", which distributed to the validators, these fees are
↪  increased in every sweep call,
↪  https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/staking/InfraredBERAFeeReceivor.sol#L77.

The other part of this is `2@`, where these fees are "claimed" by the Infrared contract, by calling
↪  `InfraredBERA::collect`, which calls `InfraredBERAFeeReceivor::collect`,
↪  https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/staking/InfraredBERA.sol#L269.

This is where the issue is, where `InfraredBERA::collect` will revert if the "shareholder fees" are below a
↪  certain threshold:
```solidity
    function collect() external returns (uint256 sharesMinted) {
        if (msg.sender != InfraredBERA) revert Errors.Unauthorized(msg.sender);
        uint256 shf = shareholderFees;
        if (shf == 0) return 0;

        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (shf < min) {
@>          revert Errors.InvalidAmount();
        }

        if (shf > 0) {
            delete shareholderFees;
            (, sharesMinted) =
                IInfraredBERA(InfraredBERA).mint{value: shf}(address(infrared));
        }
        emit Collect(address(infrared), shf, sharesMinted);
    }
~~~

This forces the whole validators updating logic to be DoSed as long as the shareholder fees are below that threshold, as `Infrared::harvestOperatorRewards` will always revert in that case.

Proof of Concept: Add the following test in `tests/unit/core/Infrared/Validators.sol`:

```
function testDoSValidators() public {
    vm.prank(infraredGovernance);
    ibera.setFeeDivisorShareholders(3);

    deal(address(receivor), 17 ether);

    ValidatorTypes.Validator[]
        memory validators = new ValidatorTypes.Validator[](1);
    validators[0] = ValidatorTypes.Validator({
        addr: address(1),
        pubkey: abi.encodePacked(address(1))
    });

    // shf = 5.6 BERA < min = 11 BERA
    vm.prank(infraredGovernance);
    infrared.addValidators(validators);
}
```

Recommendation: `src/staking/InfraredBERAFeeReceivor.sol`:

```diff
    function collect() external returns (uint256 sharesMinted) {
        if (msg.sender != InfraredBERA) revert Errors.Unauthorized(msg.sender);
        uint256 shf = shareholderFees;
        if (shf == 0) return 0;

        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (shf < min) {
-           revert Errors.InvalidAmount();
+           return 0;
        }

        if (shf > 0) {
            delete shareholderFees;
            (, sharesMinted) =
                IInfraredBERA(InfraredBERA).mint{value: shf}(address(infrared));
        }
        emit Collect(address(infrared), shf, sharesMinted);
    }
```

3.1.3 Asynchronous Reward Distribution in InfraredVault Causes Loss of Earned Rewards for Early Exiters

Severity: High Risk

Context: MultiRewards.sol#L230

Summary: The InfraredVault's reward distribution mechanism creates a timing mismatch between when rewards are earned and when they're distributed. Users who exit the protocol before a harvest cycle may lose their rightfully earned iBGT rewards, despite their tokens contributing to generating those rewards.

Finding Description: The vulnerability stems from the reward harvesting and distribution flow in the InfraredVault system:

1. Users stake their tokens in InfraredVault.
2. These staked tokens generate iBGT rewards in BerachainRewardsVault.
3. When a user calls `exit --> getReward --> getRewardForUser()`:

```
// MultiRewards.sol
function getRewardForUser(address _user) public {
    onReward(); // Triggers harvest
    // ... distribute rewards
}

// InfraredVault.sol -> RewardsLib.sol
function harvestVault(...) {
    // ... harvest logic ...
    if (_amt > 0) {
        vault.notifyRewardAmount(ibgt, _amt); // Starts new reward period
    }
}
```

Visual Timeline:

```
T0: User stakes 100 tokens
|
T1: Protocol generates rewards (not yet harvested)
|
T2: User unstakes
|
T3: harvest() occurs, rewards are notified and start distributing
|
Result: User misses rewards earned during T0-T2
```

Note that `notifyRewardAmount()` starts distributing iBGT rewards (like other rewards) from the moment it's called, rather than accounting for the period during which these rewards were actually generated. Note that this is only an issue with iBGT tokens. iBGT needs to be rewarded retro-actively for a fair distribution.

Impact Explanation: Users who exit before harvest cycles lose their earned rewards. Creates unfair advantage based on exit timing..

Likelihood Explanation: This affects all users who exit the protocol every time they exit.

Tools Used: Manual Review.

Recommendations: Implore a system that tracks historical stakes and distributes rewards based on stake weight during the actual reward generation period. You could also consider a vault shares system just for iBGT rewards. When users stake and withdraw, they own and claim their fair share of the pool iBGT tokens.

3.1.4 In `infrared::claimLostRewardsOnVault()` Lost Vault rewards are handled inappropriately

Severity: High Risk

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

Summary:

`Infrared::claimLostRewardsOnVault()` retuned tokens will be used as bribes and not split between iBERA share holders. **Finding Description:** in `Infrared::claimLostRewardsOnVault()`, we call `vault.getReward()` to get any rewards that were initially sent to the infrared staking vault with no stakers.

The problem is that when we call `vault.getReward()`, we don't do any thing after, leaving those reward tokens inside `infrared` contract.

Those tokens standing in `infrared` contract can be sent forcebly to the `BribeCollector` by calling `infrared::harvestBribes()` that will forward any whitelisted balance tokens to the collector contract as shown here.

```
File: RewardsLib.sol
    function harvestBribes(
        RewardsStorage storage $,
        address wbera,
        address collector,
        address[] memory _tokens,
        bool[] memory whitelisted
    ) external returns (address[] memory tokens, uint256[] memory _amounts) {
        uint256 len = _tokens.length;
        _amounts = new uint256[](len);
        tokens = new address[](len);

        for (uint256 i = 0; i < _tokens.length; i++) {
            if (!whitelisted[i]) continue;
            address _token = _tokens[i];
            if (_token == DataTypes.NATIVE_ASSET) {
                IWBERA(wbera).deposit{value: address(this).balance}();
                _token = wbera;
            }
            // amount to forward is balance of this address less existing protocol fees
            uint256 _amount = ERC20(_token).balanceOf(address(this))
                - $.protocolFeeAmounts[_token];
            _amounts[i] = _amount;
            tokens[i] = _token;
            _handleTokenBribesForReceiver($, collector, _token, _amount);
        }
    }
```

Those tokens sent as bribes will be auctioned (build up), till some one buy it with `wBERA` token as the `payoutToken`, and call `infrared.collectBribes` during the buying process.

```
File: BribeCollector.sol
        infrared.collectBribes(payoutToken, payoutAmount);
        // payoutAmount will be transferred out at this point

```

then `collectBribes()` calling `collectBribesInWBERA()` that again split the `wBERA` amount to infrared vault and `iBERA` holders. **Impact Explanation:** High:

- iBERA holders lose tokens that initially should have been to them.
- Lost infrared vault rewards are used as Bribes. **Likelihood Explanation:** High:
- every time a governance calls `claimLostRewardsOnVault()` it will be used as bribes by `harvest-Bribes()` split to protocol, voters, infrared vault and iBERA.

3.1.5 First validator deposit front-running in InfraredBERADepositor

Severity: High Risk

Context: InfraredBERADepositor.sol#L109-L134

Description: InfraredBERADepositor allows anyone to execute deposits after a 7-day delay period, not just the keeper. This creates a front-running vulnerability where an attacker can become the validator operator by front-running the first legitimate deposit.

```
// Current vulnerable check
if (!kpr && !_enoughtime(s.timestamp, uint96(block.timestamp))) {
    revert Errors.Unauthorized(msg.sender); // @audit after 7 days anyone can call execute?
}
```

Proof of Concept: The flow is simple, a legitimate user queues a deposit, but after the delay period an attacker front-runs the execution of this deposit, setting themselves as the operator. When the legitimate keeper attempts their deposit, it fails due to the operator mismatch. The attacker now controls validator BGT and incentive reward collection until the validator shuts down their node.

```
function testFrontrunOperatorFrontrunning() public {
    // Setup initial state
    uint256 value = InfraredBERAConstants.INITIAL_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
    vm.deal(address(ibera), value);
    // Queue legitimate deposit
    vm.prank(address(ibera));
    depositor.queue{value: value}(InfraredBERAConstants.INITIAL_DEPOSIT);

    // Setup governance signature
    vm.prank(infraredGovernance);
    ibera.setDepositSignature(pubkey0, signature0);

    // Warp to after minimum delay
    vm.warp(block.timestamp + InfraredBERAConstants.FORCED_MIN_DELAY + 1);

    // Attacker front-runs with malicious operator
    address attacker = makeAddr("attacker");
    bytes memory maliciousOperator = abi.encodePacked(attacker);

    // Mock BeaconDeposit to accept malicious operator first
    vm.mockCall(
        depositor.DEPOSIT_CONTRACT(),
        abi.encodeWithSelector(IBeaconDeposit.getOperator.selector, pubkey0),
        abi.encode(address(0))  // Mock that no operator is set yet
    );

    vm.prank(attacker);
    depositor.execute(pubkey0, InfraredBERAConstants.INITIAL_DEPOSIT);

    // Verify attacker succeeded in setting operator
    vm.mockCall(
        depositor.DEPOSIT_CONTRACT(),
        abi.encodeWithSelector(IBeaconDeposit.getOperator.selector, pubkey0),
        abi.encode(attacker)
    );

    // Now legitimate keeper tries to deposit, but fails due to wrong operator
    vm.expectRevert(Errors.UnauthorizedOperator.selector);
    vm.prank(keeper);
    depositor.execute(pubkey0, InfraredBERAConstants.INITIAL_DEPOSIT);
}
```

As a result of this, attacker can control validator POL operations. He can collect validator BGT and incentive rewards, he can also force legitimate validator to shut down and disrupts protocol operations.

Recommendation: Restrict execute() to keeper only by removing the time-based authorization:

```
function execute(bytes calldata pubkey, uint256 amount) external {
    // Only allow keeper
    if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
        revert Errors.Unauthorized(msg.sender);
    }

    // Rest of the function...
}
```

This ensures only trusted keepers can set validator operators through first deposits.

3.1.6 Users staked funds at risk due to non-enforced sequential deposits

Severity: High Risk

Context: InfraredBERADepositor.sol#L191

Description: A bug in Berachain's beacon-kit affects deposit inclusion validation (PR #2407). The issue allows validators to potentially skip including deposits in blocks, breaking the expected sequential ordering of deposits.

InfraredBERA uses Berachain's deposit contract through `InfraredBERADepositor.sol`, making it vulnerable to:

1. Synchronization failures if historical deposits were excluded.
2. State inconsistencies between InfraredBERA's accounting (`nonceSubmit`, slips) and actual on-chain state.
3. Potential blocking of new deposits if they depend on previously excluded deposits.

The main problem here is that if validators skip including deposits, users' funds sent through Infrared-BERADepositor `IBeaconDeposit(DEPOSIT_CONTRACT).deposit{value: amount}` could get stuck in limbo - not properly registered on-chain but deducted from users' balances - because our contract assumes all deposits are included sequentially but the protocol currently doesn't enforce this.

Recommendation: Add tracking for deposit inclusion confirmation:

```
// In InfraredBERADepositor.sol
function execute(bytes calldata pubkey, uint256 amount) external {
    // Add deposit inclusion verification
    require(IBeaconDeposit(DEPOSIT_CONTRACT).isDepositIncluded(pubkey), "Deposit not included");
    // Existing code...
}
```

Monitor PR #2407 and prepare for potential protocol upgrade after merge.

or maybe consider adding retry mechanism for failed deposits to handle temporary inclusion issues.

3.1.7 Deposits exceeding the maximum effective balance will be stuck

Severity: High Risk

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

Description: In the consensus layer, if a validator's balance exceeds 10M BERA, any balance excess of it will be automatically withdrawn via a partial withdrawal:

https://github.com/berachain/beacon-kit/blob/419cd6608323e01728e9a8bfc8a0f791aab8bba5/state-transition/core/state/statedb.go#L168-L185.

```
        } else if validator.IsPartiallyWithdrawable(
            balance, math.Gwei(s.cs.MaxEffectiveBalance()),
        ) {
            withdrawalAddress, err = validator.GetWithdrawalCredentials().ToExecutionAddress()
            if err != nil {
                return nil, err
            }

            withdrawals = append(withdrawals, engineprimitives.NewWithdrawal(
                math.U64(withdrawalIndex),
                validatorIndex,
                withdrawalAddress,
                balance-math.Gwei(s.cs.MaxEffectiveBalance()),
            ))

            // Increment the withdrawal index to process the next withdrawal.
            withdrawalIndex++
        }
```

The problem is that in the `InfraredBERADepositor`, when depositing BERA via `execute`, there is **no check that the validator's balance after the deposit will exceed maximum effective balance of 10M BERA** Hence, a malicious user during forced staking or even the keeper can stake BERA to a validator that is already at the maximum effective balance of 10M BERA..

This will result in the excess BERA being automatically withdrawn via a partial withdrawal to `InfraredBERAWithdraworLite`. Since `InfraredBERAWithdraworLite` only supports full withdrawals when a validator is evicted from the active set, the funds from the partial withdrawal will be stuck there.

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/staking/InfraredBERAWithdraworLite.sol#L118-L152.

```
    function sweep(bytes calldata pubkey) external {
        // only callable when withdrawals are not enabled
        if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
            revert Errors.Unauthorized(msg.sender);
        }
        // onlyKeeper call
        if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
            revert Errors.Unauthorized(msg.sender);
        }
        // Check if validator has already exited - do this before checking stake
        if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
            revert Errors.ValidatorForceExited();
        }
        // forced exit always withdraw entire stake of validator
        uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

        // do nothing if InfraredBERA deposit would revert
        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (amount < min) return;
        // revert if insufficient balance
        if (amount > address(this).balance) revert Errors.InvalidAmount();

        // todo: verfiy forced withdrawal against beacon roots

        // register new validator delta
        IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

        // re-stake amount back to ibera depositor
        IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
            value: amount
        }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

        emit Sweep(InfraredBERA, amount);
    }
```

Recommendation: Check in `InfraredBERADepositor.execute` if the validator's current stake plus the amount will exceed the maximum effective balance of 10M BERA and revert if so.

3.1.8 Insufficient initial deposit validation can cause a deposit signature mismatch on consensus layer and subsequent loss of funds

Severity: High Risk

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

Description: In this report, we will show how due to insufficient initial deposit validation, **frontrunning** plus having an **arbitrary amount be supplied** can be combined to cause a deposit signature mismatch on the consensus layer leading to a permanent loss of funds.

This arises because this frontrunning attack causes a mismatch on the EL and CL on whether an initial deposit has been made from the `InfraredBERADepositor` contract. Whereby the `InfraredBERADepositor` can be fooled into thinking that it has made an initial deposit on the EL therefore allowing an arbitrary `amount` to be made. Whereas in reality, the initial deposit has not been made yet in the CL.

Let's go through how two benign behaviours when combined can cause the attack:

1. It is well-known that the `BeaconDeposits` contract is susceptible to frontrunning to set an operator address, this can be seen in:

https://github.com/berachain/contracts/blob/main/src/pol/BeaconDeposit.sol#L105-L118.

```
        // Set operator on the first deposit.
        // zero `_operatorByPubKey[pubkey]` means the pubkey is not registered.
        if (_operatorByPubKey[pubkey] == address(0)) {
            if (operator == address(0)) {
                ZeroOperatorOnFirstDeposit.selector.revertWith();
            }
            _operatorByPubKey[pubkey] = operator;
            emit OperatorUpdated(pubkey, operator, address(0));
        }
        // If not the first deposit, operator address must be 0.
        // This prevents from the front-running of the first deposit to set the operator.
        else if (operator != address(0)) {
            OperatorAlreadySet.selector.revertWith();
        }
```

Berachain team is aware of this and recommends that validators should recognise such an attack if the initial deposit fails because of the operator address not being `address(0)`.

However, `InfraredBERADepositor` cannot recognise such a frontrunning attack if the attacker provides a correct operator. If an attacker frontruns and provides a correct operator in the deposit contract matching the Infrared operator, `InfraredBERADepositor` will treat any deposit as a subsequent deposit, and everything succeeds.

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/staking/InfraredBERADepositor.sol#L123-L133.

```
        } else {
            // Verify subsequent deposit requirements
            if (currentOperator != operator) {
                revert Errors.UnauthorizedOperator();
            }
        }
        ...
        // if operator already exists on BeaconDeposit, it must be set to zero for new deposits
        if (currentOperator == operator) {
            operator = address(0);
        }
```

2. The next behaviour we want to highlight is that an attacker can specify an arbitrary `amount` in `execute`.

```
    /// @inheritdoc IInfraredBERADepositor
    function execute(bytes calldata pubkey, uint256 amount) external {
```

However, during the initial deposit, we enforce that `amount` is equal to the `InfraredBERAConstants.INITIAL_DEPOSIT`. This is because signature verification is done in the consensus layer for only the initial deposit over the tuple (pubkey, credentials, amount). Since, we do not want the signature verification to fail `amount` is set equal to the `InfraredBERAConstants.INITIAL_DEPOSIT`.

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/staking/InfraredBERADepositor.sol#L123-L128.

```
        // Add first deposit validation
        if (currentOperator == address(0)) {
            if (amount != InfraredBERAConstants.INITIAL_DEPOSIT) {
                revert Errors.InvalidAmount();
            }
        }
```

Putting it together: Let's see how an attacker can put this together to cause a deposit signature mismatch on the CL.

1. First, the attacker frontruns by registering Infrared's pubkey with the correct operator but an incorrect signature. The `BeaconDeposit` contract on the EL will reflect the correct operator. However, on the CL, since the signature is incorrect the validator will not be created on the CL side.
2. Next, during forced staking, the malicious user can call `InfraredBERADepositor.execute` with an arbitrary amount, since a correct operator has been set on the `BeaconDeposit` contract, the amount need not be the `InfraredBERAConstants.INITIAL_DEPOSIT` but rather any amount.

3. This deposit succeeds on the EL, however since the CL does not recognise any initial deposit from the validator it will attempt to create a validator and perform signature verification. However, signature verification will fail on the CL side, as the amount does not correspond to the deposit signature, this will cause a loss of funds.

Recommendation: The problem is that the `InfraredBERADepositor` treats the `BeaconDeposit` contract as a source of truth on whether an initial deposit has been made via checking the operator address. However since anyone can frontrun and set an operator in the `BeaconDeposit` contract, it cannot be trusted.

Consider tracking in the `InfraredBERADepositor` whether an initial deposit has been made from the `InfraredBERADepositor` contract, for example instead of relying on whether a `currentOperator` has been set, such a change can be made.

```diff
        // Add first deposit validation
-       if (currentOperator == address(0)) {
+       if (!IInfraredBERA(InfraredBERA).staked(pubkey))
+           if (currentOperator != address(0)) {
+               // SHOULD REVERT HERE
+           }
            if (amount != InfraredBERAConstants.INITIAL_DEPOSIT) {
                revert Errors.InvalidAmount();
            }
```

3.1.9 Returned BERA in the WithdraworLite contract can be frozen under several conditions

Severity: High Risk

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

Summary: In the InfraredBERAWithdraworLite's contract (in scope), there's a `sweep()` function which can only be called by the keeper. The purpose of this function is to re-deposit the BERA to the Berachain's deposit contract in case if a validator has force exited..

The problem is that there is a condition which makes sure that the Bera balance of the Withdrawor contract is equal or more than the virtual balance of the force-exited validator, otherwise it reverts.

There are several situations in which the returned amount from BERA doesn't always have to resemble the virtual balance logged in InfraredBera..

- A portion of a validator's stake can be slashed if they misbehave, and if they force exit afterwards or were excluded from the validator set, the remaining BERA will be returned.
- Amounts that go over the maximum deposit that a validator can have will be returned to the address included in the credentials.

Finding Description: The Berachain validators are subject to slashing. In V1, slashing could affect both validators and BGT delegators. In V2, only validators' BERA stakes are subject to slashing. This mechanism is a punishment for validators who misbehave or perform poorly. If a validator is slashed, a portion of their staked tokens is taken..

In the WithdraworLite contract, the `sweep()` function exists to re-deposit the BERA tokens of a validator which has force exited:

```
function sweep(bytes calldata pubkey) external {
        // only callable when withdrawals are not enabled
        if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
            revert Errors.Unauthorized(msg.sender);
        }
        // onlyKeeper call
        if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
            revert Errors.Unauthorized(msg.sender);
        }
        // Check if validator has already exited - do this before checking stake
        if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
            revert Errors.ValidatorForceExited();
        }
        // forced exit always withdraw entire stake of validator
        uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

        // do nothing if InfraredBERA deposit would revert
        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (amount < min) return;
        // revert if insufficient balance
        if (amount > address(this).balance) revert Errors.InvalidAmount();

        // register new validator delta
        IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

        // re-stake amount back to ibera depositor
        IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
            value: amount
        }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

        emit Sweep(InfraredBERA, amount);
    }
```

As it can be seen from the code block above, the virtual balance of the validator registered in the InfraredBERA is loaded:

```
uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);
```

After the said balance is loaded, it's compared against the current balance of the Withdrawor, to make sure that the balance of the Withdrawor contract is equal or more than the virtual balance (the reasoning for this check is that forced exits always withdraw the entire stake of a validator):

```
if (amount > address(this).balance) revert Errors.InvalidAmount();
```

The problem with this assumption is that never takes into consideration that validator stakes can be decreased due to slashing..

As per the official Berachain documentation, a validator's Bera stakes can be slashed if they misbehave: `https://docs.berachain.com/learn/pol/participants`.

Slashing refers to the process of penalizing a validator for misbehaving which usually takes the form of a behavior such as proposing conflicting blocks.

If a validators' stake is partially slashed, following which the validator force exits the Beacon staking mechanism, the amount of Bera which will be sent to the Withdrawor (the address included in the credentials during the Bera deposit) will be the decreased amount due to the slashing event. Due to this the balance of the Withdrawor contract will be less than the virtual balance of said validator, which means that `sweep()` will always revert. This causes the Bera within the contract to remain frozen and not be able to be re-deposited into Berachain's deposit contract.

Besides the frozen Bera funds, the validator's balance won't be able to be decreased virtually and the validator will have an "active" status, while at the same time they have force exited. This also means that their pubkey can be used to make new deposits via the InfraredBeraDeposit (considering that users can also make deposits besides the keeper), while at the same time said validator is no longer an active validator in Berachain's PoL..

There's another situation in which this impact can occur. In case if a validator's stake goes above a certain threshold (10M Bera atm): `https://docs.berachain.com/learn/pol/faqs` -> the remainder will get returned to the witdrawor address as this was the one which was included in the credentials..

The above is also a cause which will lock the funds within the Withdrawor contract, as the balance of the contract won't match the virtual balance of the validator..

Impact Explanation: Due to the condition present in the `sweep` function, if a validator has been slashed (partially decreased their active stake), and has force exited, the returned Bera balance will remain stuck in the Withdrawor Lite contract.

Likelihood Explanation: Berachain slashes all validators which misbehave, and/or a malicious validator could intentionally impose such a behavior in order to cause the fund freeze.

Proof of Concept:

- Let's say that a certain validator has an active stake of 70,000 BERA deposited via the Infrared's system.
- They intentionally or unintentionally misbehave (propose a conflicting block for example), and their stake gets slashed.
- For the purpose of this exercise, let's say that 1/3 of the stake was slashed, and their active stake has decreased to 46,667 Bera.
- Validator force exits, and their stake is sent to the Withdrawor contract as that was the contract included in the credentials during the deposit.
- Keeper notices the forced exit and calls `sweep` in order to queue the balance for a re-deposit (via a new validator) and at the same time decrease the validator's virtual balance and change their status as `exited`..
- Due to the following condition, the `sweep` function will revert every time, as the virtual balance is still 70,000 Bera, while the balance of the contract is 46,667 Bera..

```
if (amount > address(this).balance) revert Errors.InvalidAmount();
```

- The Bera balance will remain frozen in the Withdrawor contract while the virtual balance of the validator, as well as their `exited` status will be wrong.

Recommendation: Remove the condition to account for edge case scenarios such as this one.

3.1.10 Validator Can Front-Run the Deposit Using Their Own Withdrawal Credentials

Severity: High Risk

Context: InfraredBERADepositor.sol#L191-L193

Summary: A malicious validator can intercept and front-run the deposit transaction with credentials under their control, effectively stealing staked funds meant for the contract. This vulnerability arises because the contract does not verify the current `BeaconDeposit` contract state or otherwise ensure that the validator's withdrawal credentials have not been preemptively set to a malicious address.

Finding Description: During the `execute` function call, there is no mechanism ensuring that the validator key has not been front-run using different withdrawal credentials. A malicious validator could deposit first, specifying their own credentials. Subsequent deposits from the `InfraredBeraDepositor` contract will then increase the balance of a validator that can ultimately withdraw to an address under the attacker's control. This breaks a core security guarantee, because the protocol loses the ability to control or retrieve the funds once a validator's withdrawal credentials are compromised.

Note: While the contest's scope doesn't cover the actual withdrawal process, this finding is referencing the setting malicious withdrawal credentials during the **deposit** process, that allows an attacker to steal funds as soon as withdrawals become enabled, even if the withdrawal contract itself is implemented correctly.

Impact Explanation: This vulnerability is assessed as **high** impact because it enables direct theft of staked BERA tokens. Once the malicious withdrawal credentials are established, there is no straightforward way for the protocol to recover or redirect the funds.

Likelihood Explanation: The likelihood is **medium** because using third-party validators without proper verification could allow malicious validators to be whitelisted, creating an opportunity for this exploit. If the protocol does not implement strict checks or continuous monitoring mechanisms, the risk of malicious validators successfully executing this attack remains significant.

Proof of Concept: While implementing a full PoC requires setting up a complete beacon node environment, the high-level attack scenario is outlined below:

1. A whitelisted validator has not yet deposited into the BeaconDeposit contract..
2. The malicious validator observes an incoming `execute` transaction with their `pubkey` and initial deposit in the mempool..
3. Before keeper finalizes the deposit, the validator submits their own deposit of 10k BERA to the `BeaconDeposit` contract, setting withdrawal credentials to a malicious address..
4. After this front-running deposit is confirmed, the malicious withdrawal credentials are now recorded for that `pubkey`, and any subsequent deposit from `InfraredBERADepositor` cannot overwrite them..
5. The validator exits the stake with these malicious credentials, effectively stealing the staked BERA.

Below are relevant references from the Berachain beacon-kit repository:

- **Minimum deposit requirement**: 10k BERA. (mainnet.go#L65).
- **Deposit processing**: The main entry point is `processDeposit`. (state_processor_staking.go#L69).
- **Retrieving the validator**: Inside `applyDeposit`, a lookup is performed. (state_processor_staking.go#L91).
- **Creating a new validator**: `createValidator` is called if no validator is found. (state_processor_staking.go#L98),. triggering `addValidatorToRegistry`. (state_processor_staking.go#L167). to build a new validator. (state_processor_staking.go#L172). with the requirement that the deposit must exceed 10k BERA. (validator.go#L99).
- **Only increasing balance if validator exists**: If a validator is found (`ValidatorIndexByPubkey` doesn't return an error), the code path. (state_processor_staking.go#L102). is taken, which simply increases the balance without altering existing `withdrawalCredentials`.

Recommendation: Verify that no prior deposit has been made to the intended `pubkey` with malicious credentials at the time of executing `InfraredBERADepositor.execute` transaction.

One strategy is to use an off-chain oracle to continuously monitor deposits made by whitelisted validators into the `BeaconDeposit` contract, ensuring no malicious initial deposits are made. The oracle should provide the last observed `depositCount` (BeaconDeposit.sol#L53), which must be cryptographically signed by either a trusted off-chain service or multiple off-chain entities whose signatures are aggregated using a BLS scheme. If a deposit containing malicious withdrawal credentials is detected, the validator should be promptly dewhitelisted, and the `depositCount` should only be provided after this dewhitelisting occurs..

During the `execute` transaction, keepers must submit both the observed `depositCount` (as verified by the oracle) and the corresponding signatures. The contract must then verify that the current `BeaconDeposit.depositCount()` matches the submitted `depositCount` and that the provided signatures are valid.

By implementing these checks, the protocol can ensure that no previous deposit has established malicious withdrawal credentials before proceeding to stake the user's BERA.

Alternatively, require that validators deposit the minimum 10k BERA with protocol-approved withdrawal credentials before being whitelisted, ensuring only authorized credentials are submitted.

3.1.11 Mismanagement of state

Severity: High Risk

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

Summary: In the InfraredBERAFeeReceivor contract within the function collect the logic doesn't seem to ensure that all preconditions for minting and fee deletion are met. If the external call does not behave as expected, deletion of shareholderfees will occur.

Finding Description: If shareholderfees is deleted and the external call to mint fails, the state is already cleared, leading to potential loss of funds.

The code to fix.

~~~
 if (shf > 0) {
            delete shareholderFees;
            (, sharesMinted) =
                IInfraredBERA(InfraredBERA).mint{value: shf}(address(infrared));```


**Impact Explanation:** The deletion of shareholderfees before a successful external call could lead to
↪  incorrectly updated state, including clearing fees that should have been tracked. this leads to
↪  inconsistent state and funds is at risk of being loss.

**Likelihood Explanation:** It has high occurrence of happening mostly due to failure to revert on an invalid
↪  state or recover from errors which will cause inconsistent state.

**Proof of Concept:** Successful External call should be made before deleting shareholderfees, in that way the
↪  state will be updated correctly.

**Recommendation:** To mitigate this issue of mismanagement of state move the delete shareholderfees statement
↪  after a successful external call.

A snippet of the fixed code.

```javascript
 if (shf > 0) {

    (, sharesMinted) =
                IInfraredBERA(InfraredBERA).mint{value: shf}(address(infrared));
                delete shareholderFees;```





### External call failure
<!--
Number: 362
Candinacode repository status: spam
Hyperlink: [Issue 362](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/findings/362)
Labels: [None]
Fixed on: [None]
-->


**Severity:** High Risk

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

**Summary:** In the contract IInfraredBERAFeeReceivor within the function collect external call is highly to
↪  occur.


**Finding Description:** The external call to IInfraredBERA(InfraredBERA).mint assumes it will succeed, but it
↪  might revert due to insufficient gas, unexpected behavior, or errors in the InfraredBera contract. if the
↪  external call fails after delete shareholderfees, the state is already cleared, leading to loss of funds or
↪  inconsistent behavior.

The code to fix.
$CODEBLOCK0javascript.
.
InfraredBERA(InfraredBERA).mint{value:shf}(address(infrared)).
returns (uint256 minted) {.
                  sharesminted=minted;

// clear shareholderfees only after a successful call.
             delete shareholderfees;
.
emit.
collect(address(infrared), shf, sharesMinted);
}catch{.

//Handle failure.
revert.
     Errors.ExternalCallFailed();
}.
   return sharesMinted;
}.
~~~

~~~
### Double Boost Queue Manipulation in Validator System
<!--
Number: 364
Candinacode repository status: new
Hyperlink: [Issue 364](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/findings/364)
Labels: [None]
Fixed on: [None]
-->


**Severity:** High Risk

**Context:** [Infrared.sol#L618](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/core/Infrar⌋
↪   ed.sol#L618),
↪   [ValidatorManagerLib.sol#L106](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/core/libr⌋
↪   aries/ValidatorManagerLib.sol#L106)

**Summary:** The `Infrared` protocol validator reward system contains an issue that allows validators to
↪   receive multiple rewards through duplicate queue entries. This can lead to unfair reward distribution and
↪   manipulation of the protocol's reward mechanism.

**Finding Description:** The current `queueBoosts` function does not check for duplicate pubkeys in the input
↪   array and does not check for previously queued pubkeys. This can lead to "double-queuing" where the same
↪   validator can get boosted more than once.
```solidity
// In ValidatorManagerLib
function queueBoosts(
    ValidatorStorage storage $,
    address bgt,
    address ibgt,
    bytes[] memory _pubkeys,
    uint128[] memory _amts
) external {
    if (_pubkeys.length != _amts.length) {
        revert Errors.InvalidArrayLength();
    }

    // Check total boosts
    uint256 _totalBoosts = 0;
    for (uint256 i = 0; i < _pubkeys.length; i++) {
        if (!$.validatorIds.contains(keccak256(_pubkeys[i]))) {
            revert Errors.InvalidValidator();
        }
        if (_amts[i] == 0) revert Errors.ZeroAmount();
        _totalBoosts += _amts[i];
    }

    // Verify against iBGT supply
    if (_totalBoosts > IInfraredBGT(ibgt).totalSupply() -
        (IBerachainBGT(bgt).boosts(address(this)) +
         IBerachainBGT(bgt).queuedBoost(address(this)))) {
        revert Errors.BoostExceedsSupply();
    }

    // Queue boosts
    for (uint256 i = 0; i < _pubkeys.length; i++) {
        IBerachainBGT(bgt).queueBoost(_pubkeys[i], _amts[i]);
    }
}
~~~

The function only validates the same array length between pubkeys and amounts, valid (registered) pubkey validator, amount is not null, and total boost does not exceed iBGT supply when calling `queueBoost` on BGT contract, there is no protection against double queueing for the same pubkey. This causes validators to get bigger boost than they should.

```
// In Infrared
function activateBoosts(bytes[] calldata _pubkeys) external {
    _validatorStorage().activateBoosts(address(_bgt), _pubkeys);
}
```

The activation process does not merge or validate accumulated upgrades per validator, so all queued upgrades can be activated.

Impact Explanation:

- Validators can receive disproportionate rewards.
- Undermines fair reward distribution.

Proof of Concept: Add this to `InfraredTest.t.sol` and run it `forge test --match-test testQueueBoostsDoubleQueueing -vvvv`.

```
function testQueueBoostsDoubleQueueing() public {
    // Setup: Add a validator first
    ValidatorTypes.Validator memory validator = ValidatorTypes.Validator({
        pubkey: "0x1234567890abcdef",
        addr: address(0x123)
    });
    ValidatorTypes.Validator[] memory validators = new ValidatorTypes.Validator[](1);
    validators[0] = validator;

    vm.startPrank(infraredGovernance);
    infrared.addValidators(validators);
    vm.stopPrank();

    // Setup: Mint BGT to Infrared contract to allow for boost queueing
    vm.startPrank(address(blockRewardController));
    bgt.mint(address(infrared), 1000 ether);
    vm.stopPrank();

    // Setup: Mint IBGT to create total supply
    vm.prank(address(infrared));
    ibgt.mint(address(this), 500 ether);

    // Create arrays with duplicate pubkeys
    bytes[] memory pubkeys = new bytes[](2);
    pubkeys[0] = validator.pubkey;
    pubkeys[1] = validator.pubkey; // Same pubkey again

    uint128[] memory amounts = new uint128[](2);
    amounts[0] = 100 ether;
    amounts[1] = 100 ether;

    // Queue boosts with duplicate pubkeys
    vm.startPrank(keeper);
    infrared.queueBoosts(pubkeys, amounts);
    vm.stopPrank();

    // Check that the total queued boost matches both amounts
    uint256 totalQueuedBoost = bgt.queuedBoost(address(infrared));
    assertEq(totalQueuedBoost, 200 ether, "Double queuing allowed - vulnerability confirmed");

    // Move blocks forward to allow activation
    vm.roll(block.number + HISTORY_BUFFER_LENGTH + 1);

    // Try to activate the boosts
    bytes[] memory activatePubkeys = new bytes[](1);
    activatePubkeys[0] = validator.pubkey;

    infrared.activateBoosts(activatePubkeys);

    // Verify the validator received double the intended boost
    uint256 actualBoost = bgt.boosts(address(infrared));
    assertEq(actualBoost, 200 ether, "Validator received double boost - vulnerability confirmed");

    // Additional check: Verify that queuedBoost is now 0 after activation
    totalQueuedBoost = bgt.queuedBoost(address(infrared));
    assertEq(totalQueuedBoost, 0, "Queued boost should be 0 after activation");
}
```

Result:

```
[77257] infrared::fallback([0x30783132333435363738393061626364656, 0x30783132333435363738393061626364656],
↪  [100000000000000000000 [1e20], 100000000000000000000 [1e20]])
```

Double queueing was successful, it can be seen that the system received two identical pubkeys with the same boost amount (100 ether each).

```
[46778] bgt::queueBoost(0x30783132333435363738393061626364656, 100000000000000000000 [1e20])
[4978] bgt::queueBoost(0x30783132333435363738393061626364656, 100000000000000000000 [1e20])
```

Both queues are accepted, the system executes queueBoost twice for the same pubkey.

```
[1762] bgt::queuedBoost(infrared: [0x03A6a84cD762D9707A21605b548aaaB891562aAb]) [staticcall]
 ← [Return] 200000000000000000000 [2e20]
```

Total queue becomes double, total `queuedBoost` becomes 200 ether (double of what it should be).

```
[145637] bgt::activateBoost(infrared: [0x03A6a84cD762D9707A21605b548aaaB891562aAb],
↪  0x30783132333435363738393061626364656)
   [67929] ERC1967Proxy::fallback(infrared: [0x03A6a84cD762D9707A21605b548aaaB891562aAb], 200000000000000000000
↪  [2e20])
```

Activation results in a double boost, upon activation, validators receive a 200 ether boost (double what they should).

```
[1789] bgt::boosts(infrared: [0x03A6a84cD762D9707A21605b548aaaB891562aAb]) [staticcall]
 ← [Return] 200000000000000000000 [2e20]
```

Final boost amount confirmed, final boost amount confirmed 200 ether, proving validators get double boost.

Recommendation: Implement duplicate pubkey check in `ValidatorManagerLib`.

```
function queueBoosts(
    ValidatorStorage storage $,
    address bgt,
    address ibgt,
    bytes[] memory _pubkeys,
    uint128[] memory _amts
) external {
    if (_pubkeys.length != _amts.length) {
        revert Errors.InvalidArrayLength();
    }

    // Track used pubkeys
    mapping(bytes32 => bool) memory usedPubkeys;
    uint256 _totalBoosts = 0;

    for (uint256 i = 0; i < _pubkeys.length; i++) {
        bytes32 pubkeyHash = keccak256(_pubkeys[i]);

        // Prevent duplicate pubkeys
        if (usedPubkeys[pubkeyHash]) {
            revert Errors.DuplicateValidator();
        }
        usedPubkeys[pubkeyHash] = true;

        if (!$.validatorIds.contains(pubkeyHash)) {
            revert Errors.InvalidValidator();
        }
        if (_amts[i] == 0) revert Errors.ZeroAmount();

        // Track total boosts
        _totalBoosts += _amts[i];
    }

    // Existing supply checks...

    // Queue boosts after all validations pass
    for (uint256 i = 0; i < _pubkeys.length; i++) {
        IBerachainBGT(bgt).queueBoost(_pubkeys[i], _amts[i]);
    }
}
```

3.1.12 Reward is updated for a user before harvesting a vault

Severity: High Risk

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

Description: Currently, the function `getRewardForUser()` calls `updateReward()` modifier that updates the reward for a particular user of the vault. The problem is that the function calls `onReward()` hook that harvests the vault and as the modifier is called before, users will lose any rewards earned after the harvesting.

Proof of Concept: Let's take a look at the “getRewardForUser()‘ functionality:

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/core/MultiRewards.sol#L225-249.

```
    function getRewardForUser(address _user)
        public
        nonReentrant
        updateReward(_user)
    {
        onReward();
        uint256 len = rewardTokens.length;
        for (uint256 i; i < len; i++) {
            address _rewardsToken = rewardTokens[i];
            uint256 reward = rewards[_user][_rewardsToken];
            if (reward > 0) {
                rewards[_user][_rewardsToken] = 0;
                (bool success, bytes memory data) = _rewardsToken.call(
                    abi.encodeWithSelector(
                        ERC20.transfer.selector, _user, reward
                    )
                );
                if (success && (data.length == 0 || abi.decode(data, (bool)))) {
                    emit RewardPaid(_user, _rewardsToken, reward);
                } else {
                    continue;
                }
            }
        }
    }
```

The function calls `updateReward()` modifier that updates reward for a user:

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/core/MultiRewards.sol#L78-90.

```
modifier updateReward(address account) {
        for (uint256 i; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];
            rewardData[token].rewardPerTokenStored = rewardPerToken(token);
            rewardData[token].lastUpdateTime = lastTimeRewardApplicable(token);
            if (account != address(0)) {
                rewards[account][token] = earned(account, token);
                userRewardPerTokenPaid[account][token] =
                    rewardData[token].rewardPerTokenStored;
            }
        }
        _;
    }
```

The problem is that this modifier is called before the function execution and the function itself calls `onReward()` hook that harvests the vault and updates reward amounts for users:

https://github.com/cantina-competitions/infrared-contracts/blob/develop/src/core/InfraredVault.sol#L110-112.

```
function onReward() internal override {
        IInfrared(infrared).harvestVault(address(stakingToken));
    }
```

This creates a situation where users will lose all their tokens earned after the harvesting due to the incorrect checking being performed when claiming rewards.

Recommendation: Consider updating rewards after the vault has been harvested.

3.1.13 RED token rewards will be wasted if token gets paused

Severity: High Risk

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

Summary: The RED token is distributed as reward token to Infrared Vaults. In case the token is paused during reward distribution, all accumulated rewards will be permanently lost.

Finding Description: The RED token is a pausable token that is added as a reward token to every Infrared vault when calling `Infrared.harvestVault()`. Upon token minting during the harvesting call, the mint is executed in a try/catch block to make sure that in case the token is paused it won't be added to the vault. If the token is not paused it is always added as reward token to the respective infrared vault:

```
function harvestVault(
        .....
        address red,
        ....
    ) external returns (uint256 bgtAmt) {
.....

try IRED(red).mint(address(this), redAmt) {
                {
                    // Check if RED is already a reward token in the vault
                    (, uint256 redRewardsDuration,,,,,) = vault.rewardData(red);
                    if (redRewardsDuration == 0) {
                        // Add RED as a reward token if not already added
 @>                       vault.addReward(red, rewardsDuration);
                    }
                }

              ....
            } catch {
                emit RedNotMinted(redAmt);
            }
.....

}
```

Problem comes if rewards get distributed while the RED token has been paused. This is the logic for claiming rewards from a vault:.

```
function getRewardForUser(address _user)
        public
        nonReentrant
        updateReward(_user)
    {
        ....
        for (uint256 i; i < len; i++) {
            address _rewardsToken = rewardTokens[i];
            uint256 reward = rewards[_user][_rewardsToken];
            if (reward > 0) {
                rewards[_user][_rewardsToken] = 0;
                (bool success, bytes memory data) = _rewardsToken.call(
                    abi.encodeWithSelector(
                        ERC20.transfer.selector, _user, reward
                    )
                );
                if (success && (data.length == 0 || abi.decode(data, (bool)))) {
                    emit RewardPaid(_user, _rewardsToken, reward);
                } else {
                    continue;
                }
            }
        }
    }
```

In case RED is paused the transfer call will fail, however since reverts are not respected, state will be updated and the loop will just continue, leaving tokens permanently locked in the vault.

The issue is even more serious for `WrappedVaults`, since they accumulate rewards for all their stakers to their address. Meaning that the scale of rewards being wasted here is significant and it would require only a single call.

Also `getRewardForUser()` can be called by anyone, allowing maliciously or by accident to call it while RED is paused - leading to permanently wasted rewards.

Impact Explanation: RED reward emissions will be permanently lost for individual stakers and wrapped vaults.

Recommendation: Consider removing the pausability of RED in this case or do not use it as reward token, due to the case explained.

3.1.14 Permanent DOS by locking all queued BERA in infrared Depositor

Severity: High Risk

Context: Infrared.sol#L549-L555, ValidatorManagerLib.sol#L39-L55, InfraredBERADepositor.sol#L109-L133, InfraredBERA.sol#L253-L262

Summary: Since Beacon depositor allows anyone to deposit to any validator(`pubkey`). And the initial deposit allows the depositor to register an operator for that `pubkey`. This bug can be leveraged to DOS the execution of queued deposits.

Finding Description:

1. An attacker can know very early before a deposit queue is executed on a particular `pubkey`. He has to either track `InfraredBERA.setDepositSignature` or `infrared.addValidators`. As soon as a `pubkey` is found, the attacker will do the below actions in the next block.
2. The attacker callsBeaconDeposit.deposit() with the `pubkey` registered in infrared. `credentials, signature, operator` other input parameters can be anything. Minimum deposit is 10000 gwei which is `$0.03` if ETH price is $3000 (assuming BERA == ETH).
3. Now in Beacon deposit contract, the `_operatorByPubKey` of the infrared's `pubkey` will return this malicious.
4. Now if this is first deposit, the `operator` has to be infrared address. But the `currentOperator` call will return non-zero malicious operator, so else block will get triggered. And it will revert because `currentOperator != operator`. Current operator is that malicious operator on left-hand side. And infrared address is on right-hand side.

This near to zero cost attack will cost the infrared depositor a permanent DOS, because its not possible for keeper or anyone to execute deposit to thatparticular `pubkey`. Maybe possible to execute deposit on a new registered `pubkey`. But the attacker will repeat the above attack, because executing a deposit will mandatorily require all `pubkey` to be registered and a signature has to be added. There's enough time and zero cost to repeat the attack.

Check BeaconDeposit.deposit().

```
    function execute(bytes calldata pubkey, uint256 amount) external {
        bool kpr = IInfraredBERA(InfraredBERA).keeper(msg.sender);
        // check if in *current* validator set on Infrared
        if (!IInfraredBERA(InfraredBERA).validator(pubkey)) {
    @>>     revert Errors.InvalidValidator();
        }
    ---- SNIP ----

        address operator = IInfraredBERA(InfraredBERA).infrared(); // infrared operator for validator
        address currentOperator =
            IBeaconDeposit(DEPOSIT_CONTRACT).getOperator(pubkey);
        // Add first deposit validation
        if (currentOperator == address(0)) {
            if (amount != InfraredBERAConstants.INITIAL_DEPOSIT) {
                revert Errors.InvalidAmount();
            }
        } else {
            if (currentOperator != operator) {
    @>>        revert Errors.UnauthorizedOperator();
            }
        }

    ---- SNIP ----

    }
```

Impact Explanation: Permanent DOS of queued BERA in the infrared depositor contract.

Likelihood Explanation: All the time. Doesn't matter if it's an initial deposit or a repeated one. As soon as a public key is registered, attack happens.

Recommendation: Allow governor to do an inital deposit of 10k gwei worth BERA, so that operator will always be infrared contract. This way attacker can't do initial deposit.

3.1.15 Removed validators are able to activate queued boosts and lock the tokens

Severity: High Risk

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

Description: When a validator is removed from Infrared, queued boosts for that validator cannot be activated because the `activateBoosts` function checks the validator's status:

```
    function activateBoosts(
        ValidatorStorage storage $,
        address bgt,
        bytes[] memory _pubkeys
    ) external {
        for (uint256 i = 0; i < _pubkeys.length; i++) {
=>          if (!$.validatorIds.contains(keccak256(_pubkeys[i]))) {
=>              revert Errors.InvalidValidator();
            }
            IBerachainBGT(bgt).activateBoost(address(this), _pubkeys[i]);
        }
    }
```

However, since `BerachainBGT::activateBoost` can be called by anyone on behalf of others, a malicious validator who has queued boosts can exploit this. They can directly call `activateBoost` on `BerachainBGT`, using `Infrared`'s address as the `user` and their own public key as the `pubkey`:

```
    /// @inheritdoc IBGT
    function activateBoost(address user, bytes calldata pubkey) external returns (bool) {
        QueuedBoost storage qb = boostedQueue[user][pubkey];
        (uint32 blockNumberLast, uint128 amount) = (qb.blockNumberLast, qb.balance);
        // `amount` must be greater than zero to avoid reverting as
        // `stake` will fail with zero amount.
        if (amount == 0 || !_checkEnoughTimePassed(blockNumberLast, activateBoostDelay)) return false;

        totalBoosts += amount;
        unchecked {
            // `totalBoosts` >= `boostees[validator]` >= `boosted[user][validator]`
            boostees[pubkey] += amount;
            boosted[user][pubkey] += amount;
            UserBoost storage userBoost = userBoosts[user];
            (uint128 boost, uint128 _queuedBoost) = (userBoost.boost, userBoost.queuedBoost);
            // `totalBoosts` >= `userBoosts[user].boost`
            // `userBoosts[user].queuedBoost` >= `boostedQueue[user][validator].balance`
            (userBoost.boost, userBoost.queuedBoost) = (boost + amount, _queuedBoost - amount);
        }
        delete boostedQueue[user][pubkey];

        IBGTStaker(staker).stake(user, amount);

        emit ActivateBoost(msg.sender, user, pubkey, amount);
        return true;
    }
```

This allows the malicious validator to stake the boost tokens for themselves on behalf of `Infrared.sol`.

Since the only way for the protocol to withdraw the boosts from the validator is through the `queueDropBoosts` function, and this function checks the validator's status on `Infrared.sol`, the call reverts with an `InvalidValidator` error:

```
            if (!$.validatorIds.contains(keccak256(_pubkeys[i]))) {
                revert Errors.InvalidValidator();
            }
```

Therefore, the protocol must add the validator back to the system to be able to call `queueDropBoosts`.

However, re-adding the validator is only possible if the validator has claimed all their rewards and has been purged. The `InfraredDistributor::add()` function checks if the validator already exists before re-adding:

```
        if (_validators[keccak256(pubkey)] != address(0)) {
            revert Errors.ValidatorAlreadyExists();
        }
```

And if the validator has unclaimed rewards, they cannot be fully removed/purged because the `purge` function will revert the transaction here:

```
        if (s.amountCumulativeLast != s.amountCumulativeFinal) {
            revert Errors.ClaimableRewardsExist();
        }
```

This results in the boost tokens being permanently locked, staked to the malicious validator on Berachain BGT.

Example scenario:

- A malicious validator holds 10e18 rewards on `InfraredDistributor.sol` and 100e18 queued boosts on Berachain BGT.
- The protocol removes the validator.
- The queued boosts on Berachain BGT mature and can be activated, but calling `Infrared.sol::activateBoost` reverts because the validator is removed.
- The malicious validator directly calls `activateBoost` on Berachain BGT with `Infrared.sol`'s address as `user` and their own public key as `pubkey`.
- The keepers's call to `queueDropBoosts` reverts because the validator is removed.

- The protocol attempts to re-add the validator so that `queueDropBoosts` can be called, but this reverts because the validator has not been purged. Purging is blocked by unclaimed rewards.
- Consequently, the boost tokens remain permanently locked and staked to the malicious validator on Berachain BGT.

Proof of Concept: To test the scenario, please add the following test case to `InfraredTest.t.sol`:

```
    function test_NinjaMaliciousBoost() public {
        ERC20 token = ERC20(address(infraredDistributor.token()));
        // 1. Add a validator to the validator set
        vm.startPrank(infraredGovernance);
        ValidatorTypes.Validator memory validator_str = ValidatorTypes.Validator({
            pubkey: "0x1234567890abcdef",
            addr: address(validator)
        });
        ValidatorTypes.Validator[] memory validators =
            new ValidatorTypes.Validator[](1);
        validators[0] = validator_str;
        bytes[] memory pubkeys = new bytes[](1);
        pubkeys[0] = validator_str.pubkey;
        infrared.addValidators(validators);
        vm.stopPrank();

        // Step 2: Vault Registration
        address[] memory rewardTokens = new address[](2);
        rewardTokens[0] = address(ibgt);
        rewardTokens[1] = address(red);
        InfraredVault vault = infraredVault;
        vm.prank(address(infrared));
        ibgt.mint(address(infraredGovernance), 100 ether);
        vm.startPrank(address(blockRewardController));
        bgt.mint(address(infrared), 101 ether);
        vm.stopPrank();
        address ibgtVault = address(infrared.ibgtVault());
        infrared.harvestBase();
        vm.startPrank(infraredGovernance);
        infrared.updateWhiteListedRewardTokens(address(wbera), true);
        vm.stopPrank();

        // Step 2: User Interaction - Staking Tokens
        address user = address(10);
        vm.deal(address(user), 1000 ether);
        uint256 stakeAmount = 1000 ether;
        vm.startPrank(user);
        wbera.deposit{value: stakeAmount}();
        wbera.approve(address(vault), stakeAmount);
        vault.stake(stakeAmount);
        vm.stopPrank();
        address vaultWbera = factory.getVault(address(wbera));
        vm.startPrank(address(blockRewardController));
        bgt.mint(address(distributor), 100 ether);
        vm.stopPrank();

        vm.startPrank(address(distributor));
        bgt.approve(address(vaultWbera), 100 ether);
        IBerachainRewardsVault(vaultWbera).notifyRewardAmount(
            abi.encodePacked(bytes32("v0"), bytes16("")), 100 ether
        );
        vm.stopPrank();

        // Step 4: Passage of Time for Rewards Distribution
        vm.warp(block.timestamp + 10 days); // Simulating 10 days for reward accrual

        // Step 5: Harvest Vault - Distributing Rewards
        vm.startPrank(address(vault));
        vault.rewardsVault().setOperator(address(infrared));

        vm.startPrank(keeper);
        infrared.harvestVault(address(wbera));
        vm.stopPrank();
        // Step 6: Calculate the amounts to queue.
        uint128[] memory amounts = new uint128[](1);
        amounts[0] = uint128(bgt.balanceOf(address(infrared))
            - bgt.queuedBoost(address(infrared)) - bgt.boosts(address(infrared));
        // Step 7: Queue the boosts.
        vm.startPrank(address(keeper));
        infrared.queueBoosts(pubkeys, amounts);
        vm.stopPrank();
        vm.roll(block.number + HISTORY_BUFFER_LENGTH + 1);

        deal(address(token), user, 10 ether);
        vm.startPrank(user);
        token.approve(address(infraredDistributor), 10 ether);
        infraredDistributor.notifyRewardAmount(10 ether);
        vm.stopPrank();

        // governance removes the validator
        vm.prank(infraredGovernance);
        infrared.removeValidators(pubkeys);

        // activate boost on Infrared.sol fails
        vm.expectRevert(Errors.InvalidValidator.selector);
        infrared.activateBoosts(pubkeys);

        // validator activates the boost directly on Berachain BGT
        bgt.activateBoost(address(infrared), pubkeys[0]);

        // keeper wants to queue a drop but the txn reverts
        vm.prank(keeper);
        vm.expectRevert(Errors.InvalidValidator.selector);
        infrared.queueDropBoosts(pubkeys, amounts);

        // governance tries to add the validator so the keeper can call queueDropBoosts
        // but the txn reverts with ValidatorAlreadyExists because the validator has unclaimed rewards
        // and he cant be purged
        vm.prank(infraredGovernance);
        vm.expectRevert(Errors.ValidatorAlreadyExists.selector);
        infrared.addValidators(validators);

        // purge will also fail because of the remaining rewards
        vm.prank(user);
        vm.expectRevert(Errors.ClaimableRewardsExist.selector);
        infraredDistributor.purge(pubkeys[0]);
    }
```

Run the test:

```
$ forge t --mt test_NinjaMaliciousBoost
```

Recommendation: You should cancel the queued boosts of the validator upon `removeValidator`.

3.1.16 Reentrancy Vulnerability in burn Function Leading to Incorrect Withdrawal Amounts

Severity: High Risk

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

Summary: The burn function in the contract is vulnerable to a reentrancy attack. When a user calls burn, the function first compounds the yield earned from EL rewards, which can increase the deposits value. However, the deposits value is used to calculate the withdrawal amount before the user's shares are burned. This allows an attacker to reenter the burn function multiple times, inflating the deposits value and withdrawing more funds than they are entitled to.

Check POC.

Impact Explanation: Incorrect Withdrawal Amounts: An attacker can exploit the reentrancy vulnerability to withdraw more funds than their share of the total supply.

Fund Drain: Repeated reentrant calls to burn can lead to a significant drain of funds from the contract, as the deposits value is inflated without updating the shares.

Loss of User Funds: Legitimate users may lose funds as the contract's balance is drained by the attacker.

The root cause is that it calculates first the amount and then does the external calls.

Proof of Concept: User 2 calls burn with 10 shares.

Before deposits is updated, the compound function is called, increasing deposits (e.g., by 10 ETH per call).

User 2 reenters the burn function multiple times (e.g., 3 times).

Each reentrant call to burn triggers compound, further increasing deposits:

After 1st call: deposits = 110 ETH.

After 2nd call: deposits = 120 ETH.

After 3rd call: deposits = 130 ETH.

The withdrawal amount is calculated based on the inflated deposits (130 ETH) but without updating the shares.

User 2 withdraws more funds than they should because the amount is calculated using the inflated deposits.

Recommendation: Add reenterancy guard.

3.2 Medium Risk

3.2.1 Incorrect check upon purging a validator leads to an incorrect revert

Severity: Medium Risk

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

Description: Upon purging a validator, we have the following check:

```
 if (s.amountCumulativeLast != s.amountCumulativeFinal) {
            revert Errors.ClaimableRewardsExist();
        }
```

The goal of the check is to disallow purging a validator that still has rewards to claim. However, the check is incorrect as upon adding a validator, the final cumulative value is set to 0 and his last cumulative value is set to the current one:

```
s.amountCumulativeLast = _amountsCumulative;
s.amountCumulativeFinal = 0;
```

This means that if a purge occurs when rewards still have not accrued for him, we will revert as the cumulative value will not be 0 - that is clearly incorrect as he has no rewards to claim. **Recommendation:** Refactor the check to also consider the case where the final cumulative value is 0 and the validator's last cumulative value is equal to the current cumulative value.

3.2.2 Issue with `whenNotPaused` Modifier in notifyRewardAmount Causing Paused Vaults to Prevent Reward Accumulation

Severity: Medium Risk

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

Description:: The issue arises from the use of the `whenNotPaused` modifier on the `notifyRewardAmount` function in the contract. The `whenNotPaused` modifier prevents certain functions from executing when the contract is paused. Specifically:

- The `stake` function correctly uses `whenNotPaused`, ensuring that new deposits are not accepted when the contract is paused.
- However, the `notifyRewardAmount` function, which is responsible for notifying the system about new reward tokens, also uses `whenNotPaused`.

This creates a conflict because if the vault is paused (preventing new deposits or stakes), any attempts to add rewards via `notifyRewardAmount` are also halted. This prevents the accrual of rewards, even though the vault may still accumulate new rewards for existing users or handle rewards associated with external mechanisms (like the Infrared or BERA systems). Specifically, rewards related to other tokens like BGT or rewards from `HarvestVault`, bribe collection, or `harvestBoostRewards` would not be able to propagate into the contract if it is paused.

This is problematic because even though deposits are halted, the vault still needs to continue receiving rewards, updating its reward distribution, and making reward claims. The pause mechanism effectively blocks reward accumulation, halting the flow of rewards into the vault and stopping accrual of rewards for users.

The List of functions dependent on `NotifyReward`.

1. `HarvestVault`.
2. `collectBribesInWBERA`.
3. `harvestBoostRewards`.
4. `AddIncentives`. ,result in Reverting all of these dependent function execution when notifyReward will be called.

Impact:

1. Rewards won't accumlate for vault stakers.
2. dependent function will revert.

Proof of Concept:: To illustrate this problem, consider the following scenario:

1. **Vault is Paused**:
    - The protocol admin decides to pause the vault, preventing users from making new deposits or stakes.
2. `notifyRewardAmount` **Function Call**:
    - External systems (e.g., Infrared, BERA, or other contracts) call the `notifyRewardAmount` function to add new rewards to the vault. This includes rewards like BGT, WBERA, and other incentives.
    - However, since the contract is paused (due to the `whenNotPaused` modifier), the `notifyRewardAmount` function cannot execute, and the rewards are not added.
3. **Consequences**:
    - Users who are still staked in the contract or have existing balances will not accrue rewards because the reward rate is not updated and new rewards cannot be added to vault.
    - Bribe collection, reward harvests, or other external reward injections are blocked, leading to missed opportunities for users and stopping the vault from receiving rewards.
4. **Reward Accumulation Stops**:
    - Even though users are still holding stakes and should be receiving rewards, those rewards are not being distributed or accumulated because the vault is paused and `notifyRewardAmount` cannot be called.

Recommendation:: To address this issue and allow reward accumulation to continue even when the vault is paused, you should **remove the `whenNotPaused` modifier from the `notifyRewardAmount` function** The idea is that while the contract may be paused to prevent new deposits or stakes, reward notifications should still be processed to ensure ongoing reward accumulation. The contract should allow rewards to be added to the system regardless of the pause state.

Proposed Fix::

1. **Modify the `notifyRewardAmount` function** to allow reward notifications even if the contract is paused:

```
function _notifyRewardAmount(address _rewardsToken, uint256 reward)
    internal
    updateReward(address(0)) // No `whenNotPaused` modifier here
{
    // Handle the transfer of reward tokens via `transferFrom`
    ERC20(_rewardsToken).safeTransferFrom(msg.sender, address(this), reward);
    // Add the prior residual amount and account for the new residual
    reward = reward + rewardData[_rewardsToken].rewardResidual;
    rewardData[_rewardsToken].rewardResidual = reward % rewardData[_rewardsToken].rewardsDuration;
    reward = reward - rewardData[_rewardsToken].rewardResidual;

    if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
        rewardData[_rewardsToken].rewardRate = reward / rewardData[_rewardsToken].rewardsDuration;
    } else {
        uint256 remaining = rewardData[_rewardsToken].periodFinish - block.timestamp;
        uint256 leftover = remaining * rewardData[_rewardsToken].rewardRate;

        // Calculate total and its residual
        uint256 totalAmount = reward + leftover + rewardData[_rewardsToken].rewardResidual;
        rewardData[_rewardsToken].rewardResidual = totalAmount % rewardData[_rewardsToken].rewardsDuration;

        // Remove residual before setting rate
        totalAmount = totalAmount - rewardData[_rewardsToken].rewardResidual;
        rewardData[_rewardsToken].rewardRate = totalAmount / rewardData[_rewardsToken].rewardsDuration;
    }

    rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
    rewardData[_rewardsToken].periodFinish = block.timestamp + rewardData[_rewardsToken].rewardsDuration;
    emit RewardAdded(_rewardsToken, reward);
}
```

3.2.3 Missing slippage checks from ERC4626 functions

Severity: Medium Risk

Context: WrappedVault.sol#L84-L100

Description: The WrappedVault contract, which inherits from ERC4626, uses its base deposit and withdrawal functions without implementing any slippage protection. The core ERC4626 functions (`deposit`, `withdraw`, `mint`, `redeem`) are inherited directly & do not include any mechanisms to protect users from receiving fewer shares/assets than expected due to market fluctuations.

Specifically:

1. When users deposit assets, they have no guarantee on minimum shares they'll receive.
2. When users withdraw assets, they have no guarantee on minimum assets they'll receive.
3. These transactions could be sandwiched, leading to users receiving unfavorable exchange rates.

Current implementation:

```
// WrappedVault inherits these functions from ERC4626 without modification
function deposit(uint256 assets, address receiver) public virtual returns (uint256 shares) {
    // No slippage check @audit-poc-panther
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES");
    ...
}

function withdraw(
    uint256 assets,
    address receiver,
    address owner
) public virtual returns (uint256 shares) {
    // No slippage check @audit-poc-panther
    shares = previewWithdraw(assets);
    ...
}
```

Recommendation: Modify the WrappedVault contract to override ERC4626's base functions and add slippage protection parameters.

3.2.4 The harvestBase function may fail and revert in some cases

Severity: Medium Risk

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

Summary: Before adding, removing, or replacing `Infrared validators`, the **harvestBase** function is executed..

However, this function may fail and revert in certain cases..

Consequently, these operations can also be reverted, requiring additional governance actions to resolve the issue. **Finding Description:** Managing `Infrared validators` is a crucial aspect of the protocol. Adding, removing, or replacing `Infrared validators` are governance operations, and ensuring their smooth execution is essential..

For instance, the **harvestBase** function is called during the process of adding `Infrared validators`. . https://github.com/cantina-competitions/infrared-contracts/blob/5136db07d32faaf1cab88c5e4d1a2b1ee525f262/src/core/Infrared.sol#L553.

```
function addValidators(ValidatorTypes.Validator[] calldata _validators)
    external
    onlyGovernor
{
@->    harvestBase();

    harvestOperatorRewards();
    _validatorStorage().addValidators(address(distributor), _validators);
    emit ValidatorsAdded(msg.sender, _validators);
}
```

The **harvestBase** function must not fail or revert under any circumstances.

This function calculates the difference between the `BGT balance` of `Infrared` and the `total supply` of `iBGT`. If the difference is greater than `0`, it triggers the `redeem` function in the `BGT` contract..

https://github.com/cantina-competitions/infrared-contracts/blob/5136db07d32faaf1cab88c5e4d1a2b1ee525f26
src/core/libraries/RewardsLib.sol#L124.

```
function harvestBase(address bgt, address ibgt, address ibera)
    external
    returns (uint256 bgtAmt)
{
    uint256 minted = IInfraredBGT(ibgt).totalSupply();
    uint256 bgtBalance = _getBGTBalance(bgt);
    if (bgtBalance < minted) revert Errors.UnderFlow();

    bgtAmt = bgtBalance - minted;
    if (bgtAmt == 0) return 0;

@->    IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);
}
```

The `redeem` function performs two key checks:

https://github.com/berachain/contracts/blob/4dba5adb74a5b4f7ebfb955fd7b0c8a4820a87b5/src/pol/BGT.sol#L368-L369.

```
function redeem(
    address receiver,
    uint256 amount
)
    external
@->    invariantCheck
@->    checkUnboostedBalance(msg.sender, amount)
{

}
```

1. **Unboosted Balance Check**: The `unboosted balance` of `Infrared` must be greater than or equal to the amount being redeemed.

https://github.com/berachain/contracts/blob/4dba5adb74a5b4f7ebfb955fd7b0c8a4820a87b5/src/pol/BGT.sol#L449.

```
function _checkUnboostedBalance(address sender, uint256 amount) private view {
    if (unboostedBalanceOf(sender) < amount) NotEnoughBalance.selector.revertWith();
}
function unboostedBalanceOf(address account) public view returns (uint256) {
    UserBoost storage userBoost = userBoosts[account];
    (uint128 boost, uint128 _queuedBoost) = (userBoost.boost, userBoost.queuedBoost);
    return balanceOf(account) - boost - _queuedBoost;
}
```

2. **Invariant Check**: The `BGT contract` must hold a `BERA balance` greater than or equal to its `total supply`.

https://github.com/berachain/contracts/blob/4dba5adb74a5b4f7ebfb955fd7b0c8a4820a87b5/src/pol/BGT.sol#L461.

```
function _invariantCheck() private view {
    if (address(this).balance < totalSupply()) InvariantCheckFailed.selector.revertWith();
}
```

The **harvestBase** function can revert in below two specific scenarios:

1. **User-Induced Reversion**: Any user can cause the function to revert by burning `iBGT`..

While the issue of `burning iBGT` was previously documented in the `Cantina report` under section `3.2.1` (`Burned iBGT is reminted as rewards and may lead to double-counting`), this case stems from a different root cause.

`BGT rewards` are transferred to `Infrared`, and `stakers` receive `iBGT rewards` in return. These `BGT` tokens can then be used to `boost Infrared validators`..

https://github.com/cantina-competitions/infrared-contracts/blob/5136db07d32faaf1cab88c5e4d1a2b1ee525f26 src/core/libraries/ValidatorManagerLib.sol#L173.

```
function activateBoosts(
    ValidatorStorage storage $,
    address bgt,
    bytes[] memory _pubkeys
) external {
    for (uint256 i = 0; i < _pubkeys.length; i++) {
        if (!$.validatorIds.contains(keccak256(_pubkeys[i]))) {
            revert Errors.InvalidValidator();
        }
        IBerachainBGT(bgt).activateBoost(address(this), _pubkeys[i]);
    }
}
```

When `BGT` tokens are `boosted`, the `unboosted balance` of `Infrared` decreases.

This creates a potential vulnerability where a malicious user can burn their `iBGT` to artificially increase the difference between the `BGT balance` of `Infrared` and the `total supply` of `iBGT`, exceeding the current `unboosted BGT balance` of `Infrared`..

When this happens, operations such as adding, removing, or replacing `Infrared validators` will fail and revert.

To resolve this issue, `governance` must intervene to reduce the `boosted BGT` amounts allocated to `Infrared validators`. However, this requires following the standard `governance` process, which can introduce delays.

2. The `BGT` contract does not have sufficient `BERA balance`, causing the `invariant check` to fail.

Recommendation:

```
function harvestBase(address bgt, address ibgt, address ibera)
    external
    returns (uint256 bgtAmt)
{
-    IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);

+    try IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt) {
+    } catch {
+    }
}
```

3.2.5 Malicious users can force-exit a validator forfeiting their ability to make any deposits

Severity: Medium Risk

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

Summary: If a validator's stake reaches 0, their status will be marked as `exited`, which means that no deposits can be made on their behalf..

This scenario is encompassed around a situation in which there are multiple validators and assumes that withdrawals are enabled.

Since the keeper isn't the only entity that can execute withdrawals, as this can also be done by an arbitrary user if a withdrawal is pending for more than 7 days, a malicious or just an unbeknownst user can purposefully withdraw all of the pending balance(s) from a single validator. Due to rebalancing not being atomic and no restrictions being imposed on the registration of validator balances, they are able to force exit and "remove" a validator forcefully..

Finding Description: In a situation in which there are multiple validators, the keeper/users can pick whatever validator pubkey (if the validator is approved), to deposit on behalf of.

Any inconsistencies between the balances of the validators can be rebalanced by the keeper in order to make sure that validators have equal or near equal balances during all times.

```
        IBeaconDeposit(DEPOSIT_CONTRACT).deposit{value: amount}(
            pubkey, credentials, signature, operator
        );

        // register update to stake
        IInfraredBERA(InfraredBERA).register(pubkey, int256(amount)); // safe as max fits in uint96
```

In the IBERA's `register` function is where all validator balances are being tracked:

```
function register(bytes calldata pubkey, int256 delta) external {
        if (msg.sender != depositor && msg.sender != withdrawor) {
            revert Errors.Unauthorized(msg.sender);
        }
        if (_exited[keccak256(pubkey)]) {
            revert Errors.ValidatorForceExited();
        }
        // update validator pubkey stake for delta
        uint256 stake = _stakes[keccak256(pubkey)];
        if (delta > 0) stake += uint256(delta);
        else stake -= uint256(-delta);
        _stakes[keccak256(pubkey)] = stake;
        // update whether have staked to validator before
        if (delta > 0 && !_staked[keccak256(pubkey)]) {
            _staked[keccak256(pubkey)] = true;
        }
        // only 0 if validator was force exited
        if (stake == 0) {
            _staked[keccak256(pubkey)] = false;
            _exited[keccak256(pubkey)] = true;
        }

        emit Register(pubkey, delta, stake);
    }
```

The project has also introduced a "rebalancing" functionality to make sure that they can balance out any inconsistencies or big differences between different validator balances:

```
  function queue(address receiver, uint256 amount)
        external
        payable
        returns (uint256 nonce)
    {
        bool kpr = IInfraredBERA(InfraredBERA).keeper(msg.sender);
        address depositor = IInfraredBERA(InfraredBERA).depositor();
        // @dev rebalances can be queued by keeper but receiver must be depositor and amount must exceed
        ↪  deposit fee
        if (msg.sender != InfraredBERA && !kpr) {
            revert Errors.Unauthorized(msg.sender);
        }
        if ((kpr && receiver != depositor) || (!kpr && receiver == depositor)) {
            revert Errors.InvalidReceiver();
        }
if (
            (receiver != depositor && amount == 0)
                || (
                    receiver == depositor
                        && amount <= InfraredBERAConstants.MINIMUM_DEPOSIT_FEE
                ) || amount > IInfraredBERA(InfraredBERA).confirmed()
        ) {
            revert Errors.InvalidAmount();
        }
```

```
function process() external {
        uint256 nonce = nonceProcess;
        address depositor = IInfraredBERA(InfraredBERA).depositor();
        Request memory r = requests[nonce];


        if (r.receiver == depositor) {
            // queue up rebalance to depositor
            rebalancing -= amount;
            IInfraredBERADepositor(r.receiver).queue{value: amount}(
                amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE
            );
```

The problem arises during withdrawals, if the `FORCED_MIN_DELAY` passes for a certain request, and said request can be executed by an arbitrary user, and they can use this to force exit a certain validator.

As it can be seen from the Withdrawor's `execute` function:

```
 if (!kpr && !_enoughtime(r.timestamp, uint96(block.timestamp))) {
                revert Errors.Unauthorized(msg.sender);
            }


...

 (bool success,) = WITHDRAW_PRECOMPILE.call{value: fee}(encoded);
        if (!success) revert Errors.CallFailed();

....

        // register update to stake
        IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount)); // safe as max fits in uint96
```

Once the amount is queued for withdrawals, it will be registered:

```
   function register(bytes calldata pubkey, int256 delta) external {
        if (msg.sender != depositor && msg.sender != withdrawor) {
            revert Errors.Unauthorized(msg.sender);
        }
        if (_exited[keccak256(pubkey)]) {
            revert Errors.ValidatorForceExited();
        }
        // update validator pubkey stake for delta
        uint256 stake = _stakes[keccak256(pubkey)];
        if (delta > 0) stake += uint256(delta);
        else stake -= uint256(-delta);
        _stakes[keccak256(pubkey)] = stake;
        // update whether have staked to validator before
        if (delta > 0 && !_staked[keccak256(pubkey)]) {
            _staked[keccak256(pubkey)] = true;
        }
        // only 0 if validator was force exited
        if (stake == 0) {
            _staked[keccak256(pubkey)] = false;
            _exited[keccak256(pubkey)] = true;
        }

        emit Register(pubkey, delta, stake);
    }
```

Since the Validator's stake will be 0, their `exited` status will be marked as true, and they will no longer be able to acquire positive deposit balances:

```
 if (_exited[keccak256(pubkey)]) {
            revert Errors.ValidatorForceExited();
        }
```

Impact Explanation: This will render the validator not being able to make any future deposits into Berachain.

Likelihood Explanation: Any arbitrary user can cause this to happen once the `FORCED_MIN_DELAY` passes.

Proof of Concept: Let's imagine the following hypothetical scenario: Validator Balances:

- Validator 1: 22_000e18.
- Validator 2: 21_000e18.
- Validator 3: 22_000e18.
- Validator 4: 20_000e18.

After the `FORCED_MIN_DELAY` has passed, a user decides to execute pending withdrawal(s) in the amount of 21_000e18..

Instead of initiating the withdrawal batches from multiple validators in order to decrease the amounts equally, they decide to withdraw the balance only from Validator 2.

This will cause the Validator 2's balance to be decreased to 0, and their status changed as `exited`. Once a validator's pubkey has been marked as `exited`, no deposits would be able to be made on their behalf, rendering them "useless"..

Recommendation: In case of multiple validators, enforce a mechanism in which the deposit/withdrawal amounts are deposited/taken in a "balanced" way to / out of all of the validators, instead of allowing arbitrary users to use only 1 both during deposits and withdrawals.

3.2.6 Victims could end up having to double pay fees

Severity: Medium Risk

Context: InfraredBERADepositor.sol#L159-L165

Vulnerability Details: In `InfraredBERADepositor.sol`'s `queue` function, users depositing by calling `queue` are required to provide a `fee` as well as `amount` that they wish to deposit. The `fee` provided will be used to cover the gas spent by the eventual user calling `execute` to deposit it into `IBeaconDeposit(DEPOSIT_-CONTRACT)`.

After `FORCED_MIN_DELAY`, users are allowed to force stake into infrared validators, hence the restriction that only keepers can call execute is lifted and **now any user** can call execute to deposit and receive the `fee` by the user that called `queue` previously to recompensate execute's deposit.

That is not a problem if users are genuinely helping the user at the front of the queue force stake, however the way the recompensation `fee` is released allows for attacks.

```
function execute(bytes calldata pubkey, uint256 amount) external {
    ....
    while (remaining > 0) {
        ....

        // first time loop ever hits slip dedicate fee to this call
        // @dev for large slip requiring multiple separate calls to execute, keeper must front fee in
        ↪  subsequent calls
        // @dev but should make up for fronting via protocol fees on size
        if (s.fee > 0) {
            fee += s.fee;
            s.fee = 0;
        }

        ...
    }
    ...
}
```

As shown, the fee is released the first time the loop hits the slip which means it doesn't require that the full amount is deposited.

Impact: Hence, suppose there is an user at the front of the queue with `fee: X`, `amount: 20 ether`.

Then an attacker can call execute with `amount = 1 gwei`.

- Then `execute` transfers fee X to the attacker and deposits `1 gwei` out of the `20 ether` into `IBeacon-Deposit(DEPOSIT_CONTRACT)`.

Victim's loss will hence be:

- Victim still has `20 ether - 1 gwei` undeposited, that is **equivalent to having nothing deposited** and now if the victim were to call execute to force deposit their own slip, the victim will have to bare the full cost of fee, despite having **already paid** for it during `queue` (as they are forced to provide `MINIMUM_DEPOSIT_FEE` to cover gas).. . **This means victim ends up double paying fees.**

Recommendation: Either only release fee when the whole amount is being included in the deposit, or maybe check that a significant % of amount requested by user is included before releasing fee to recompensate :).

3.2.7 The initial deposit of a new validator could fail due to the 10k ether deposit requirement and the number of slips

Severity: Medium Risk

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

Summary: Currently the contract logic in InfraredBERADepositor checks if the initial deposit is higher than 10K ether (Berachain allows validator deposits higher than 1 Ether), while at the same time imposing a minimum deposit of 10 ether within Infrared's system. It's entirely plausible to have 1000 different slips that need to be iterated over in order to reach the 10k minimum deposit cap for the initial deposit which needs to be forwarded to Berachain's deposit contract..

The problem is that this can surpass the block gas limit and result in the inability to deposit said funds to Bera, locking them in the InfraredBERADepositor.

Finding Description: If we take a look at the Berachain's Beacon Deposit Contractt, more specifically the `deposit()` function which is called in order to forward the BERA deposits from Infrared to Berachain, as seen here:

```
bytes memory credentials = abi.encodePacked(
            ETH1_ADDRESS_WITHDRAWAL_PREFIX, // 0x01
            uint88(0),
            withdrawor
        );

        if (currentOperator == operator) {
            operator = address(0);
        }
        IBeaconDeposit(DEPOSIT_CONTRACT).deposit{value: amount}(
            pubkey, credentials, signature, operator
        );
```

The `deposit()` function on Berachain's deposit contract:

```
    function deposit(
        bytes calldata pubkey,
        bytes calldata credentials,
        bytes calldata signature,
        address operator
    )
        public
        payable
        virtual
    {
        if (pubkey.length != PUBLIC_KEY_LENGTH) {
            revert InvalidPubKeyLength();
        }


        if (credentials.length != CREDENTIALS_LENGTH) {
            revert InvalidCredentialsLength();
        }


        if (signature.length != SIGNATURE_LENGTH) {
            revert InvalidSignatureLength();
        }


        // Set operator on the first deposit.
        // zero `_operatorByPubKey[pubkey]` means the pubkey is not registered.
        if (_operatorByPubKey[pubkey] == address(0)) {
            if (operator == address(0)) {
                revert ZeroOperatorOnFirstDeposit();
            }
            _operatorByPubKey[pubkey] = operator;
            emit OperatorUpdated(pubkey, operator, address(0));
        }
        // If not the first deposit, operator address must be 0.
        // This prevents from the front-running of the first deposit to set the operator.
        else if (operator != address(0)) {
            revert OperatorAlreadySet();
        }


        uint64 amountInGwei = _deposit();


        if (amountInGwei < MIN_DEPOSIT_AMOUNT_IN_GWEI) {
            revert InsufficientDeposit();
        }


        // slither-disable-next-line reentrancy-benign,reentrancy-events
        emit Deposit(
            pubkey, credentials, amountInGwei, signature, depositCount++
        );
    }
```

From the above we can see that all deposits, irrelevant whether it's the first deposit or no, must be equal to or more than the `MIN_DEPOSIT_AMOUNT_IN_GWEI` (This is checked after the 18-decimal amount was normalized to GWEI in the `_deposit()` function).

Here's the `MIN_DEPOSIT_AMOUNT_IN_GWEI` as well:

```
uint64 internal constant MIN_DEPOSIT_AMOUNT_IN_GWEI = 1e9;
```

This represents a 1 ether requirement. As seen in Infrared's conditions, all deposits must amount to at least a 10 ether value (excluding the 1 ether fee):

```
uint256 public constant MINIMUM_DEPOSIT = 10 ether;

uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT;
        fee = InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (value < min + fee) revert Errors.InvalidAmount();
```

Considering that the initial deposit must be at least 10_000 ether:

```
 if (currentOperator == address(0)) {
            if (amount != InfraredBERAConstants.INITIAL_DEPOSIT) {
                revert Errors.InvalidAmount();
            }
```

It's entirely plausible that the amount that needs to be forwarded by the first validator or any subsequent new ones can be compromised of 1000 slips (or less, but still a large amount).

The second point to this argument is the fact that deposits slips are processed in a nonce-order (i.e. they can't be skipped, canceled, etc.) and must be executed in a FCFS order..

Since the `execute()` logic in the InfraredBERADepositor iterates over all of them as well as writes to storage, each iteration can cost up to 30_000 gas, which could surpass the 30 million Berachain block gas limit at 1000 orders.

The above estimation is very conservative, excludes the other contract logic being executed, which would likely lower the amount of orders to a bit above 600-700.

```
uint256 remaining = amount;
        while (remaining > 0) {
            nonce = nonceSubmit;
            Slip memory s = slips[nonce];
            if (s.amount == 0) revert Errors.InvalidAmount();

            // @dev allow user to force stake into infrared validator if enough time has passed
            // TODO: check signature not needed (ignored) on second deposit to pubkey (think so)
            if (!kpr && !_enoughtime(s.timestamp, uint96(block.timestamp))) {
                revert Errors.Unauthorized(msg.sender);
            }

            // first time loop ever hits slip dedicate fee to this call
            // @dev for large slip requiring multiple separate calls to execute, keeper must front fee in
            ↪  subsequent calls
            // @dev but should make up for fronting via protocol fees on size
            if (s.fee > 0) {
                fee += s.fee;
                s.fee = 0;
            }

            // either use all of slip amount and increment nonce if remaining > slip amount or use remaining
            // not fully filling slip in this call
            uint256 delta = remaining > s.amount ? s.amount : remaining;
            s.amount -= delta;
            if (s.amount == 0) nonceSubmit++;
            slips[nonce] = s;

            // always >= 0 due to delta ternary
            remaining -= delta;
        }
```

If we look only at storage reads/writes, the total gas cost for each iteration (conservative) will amount up to:

- Read nonceSubmit → 2,100 gas.
- Read slips[nonce] → 2,100 gas.
- Write slips[nonce] → 20,000 gas. (Writing from a non-zero to zero value costs 5,000 but this amount is refunded after the execution of the call ends, so we're accounting it as 20k gas, as gas accounting will consider it a 20k operation).
- nonceSubmit increase (write to storage) -> 20,000 gas.

Excluding memory operations as well as other things, one iteration will conservatively cost 44k gas.

The block gas limit of Berachain is capped at 30 million. If we take into account 44k gas per iteration (excluding other operation costs), this would account for 681 iteration before the block gas limit is reached, in reality it will likely be on the lower end of 600 as we have to account for other operation costs initiated when `execute()` is called, as well as external calls and storage writes/reads in those..

With all of the above, there are two aspects to this problem.

- First and foremost is the gas griefing aspect and the fact that the Keeper will likely incur very large gas costs with each slip that it iterates over.
- The second aspect is the primary topic of this report, and the plausibility of a new validator's initial deposit to be DoSd due to the large amount of slips that need to be processed in order to account for the 10k Ether.

The second aspect can be both intentional or unintentional.. Meaning that the accumulation of a lot of slips with minimal amounts causes the inability of the validator's initial deposit to be processed can both be planned or happen coincidentally..

There are multiple other hypothetical scenarios which could cause a more critical impact.

For example, the current validator has misbehaved and needs to be removed or has force-exited. A new validator was added and an initial deposit of 10k ether is needed in order to "activate" it.

Currently there are a lot of large number of slips in the queue, since they are processed on a first-come-first-serve basis, i.e. by their nonce, 600+ slips need to be iterated over in order to reach the 10k value, BUT since the operations become too costly, the call fails due to reaching the block gas limit and the system is permanently DoSd -> considering that there's no way to cancel/refund slips. There's also an intentional aspect to this, as a malicious user could "spam" the system with hundreds of small deposits/slips in order to permanently DoS it.

Considering the above scenario, if there's no other validator which can be used to process the pending deposits, and there's no way to cancel and refund them, they would remain stuck in the Infrared system.

Although this report focuses on the 30 million block gas limit, it's likely that this transaction would fail way before reaching it since a single transaction realistically cannot use the entire 30 million gas block limit.

- While the block limit is 30 million gas, it is shared among multiple transactions, so a single transaction consuming the full limit would leave no room for others, i.e. validators would have to include one transaction per block..
- Transactions consuming such a high amount of gas could exceed practical block production time-frames, causing network delays. (Considering that Berachain's block time is 2 seconds, the network can process up to 15 million gas units per second).
- Validators might reject such transactions, especially if there's competition for block space, as a single massive transaction limits their earnings from including multiple smaller transactions.

With all of the above, realistically a lot less than 600 transactions would be needed in order to DoS the validator in processing their initial deposit and cause harm to the system.

Impact Explanation: The "activation" of a new validator could be DoSd either intentionally or unintentionally by a large number of slips due to gas consumption / block stuffing, making it impossible for the validator to be initiated within Berachain's system and/or even freeze any funds contained within InfraredBERADepositor.

Likelihood Explanation: Although an intentional attack such as this one is plausible, it's highly expensive. In parallel to this, the unintentional aspect of this taking place, still remains..

Proof of Concept: The minimalistic PoC provided below can be utilized in Remix so that gas costs for storage writes can easily be inspected.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract POC {

uint256 public amount;
uint256 public nonce;

function initialize() public {
    amount = 10e18;
    nonce = 1;
}

function estimateGasConservatively() public {
    uint256 _amount = amount;
    uint256 _nonce = nonce;

    _amount = 0;
    amount = _amount;
    _nonce++;
    nonce = _nonce;

}

}
```

Recommendation: There's no simple solution to this that would result in maintaining the minimum deposit of 10 ether..

- A possibility would be a separate flow in which any number of slips can be processed and their balance saved in a separate variable, so after they are processed independently, the keeper can execute it in a separate flow..

3.2.8 POL bribes cannot be auctioned in `BribeCollector` if too many reward tokens have already been added

Severity: Medium Risk

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

Summary:

`BribeCollector::claimFees()` always reverts if the maximum number of reward tokens has already been added to the ibgtVault. As a result, POL bribes cannot be auctioned.

Finding Description: Inside of `BribeCollector::claimFees()`, `Infrared::collectBribes()` is called to distribute the proceeds of the auction to validators, iBGT stakers, and the protocol. This function makes sure that the `payoutToken` is WBERA and calls `RewardsLib::collectBribesInWBERA()`. Now, the proportion of the bribe amount designated for InfraredBERA and the ibgtVault are determined. At the end of the call flow, `RewardsLib::_handleTokenRewardsForVault()` is called to calculate the fees and notify the ibgtVault of new rewards. However, if WBERA is not already a reward token of the ibgtVault, it first needs to be added.

```
function _handleTokenRewardsForVault(
    RewardsStorage storage $,
    IInfraredVault _vault,
    address _token,
    address voter,
    uint256 _amount,
    uint256 _feeTotal,
    uint256 _feeProtocol,
    uint256 rewardsDuration
) internal {
    if (_amount == 0) return;

    // add reward if not already added
    (, uint256 _vaultRewardsDuration,,,,,) = _vault.rewardData(_token);
    if (_vaultRewardsDuration == 0) {
>>      _vault.addReward(_token, rewardsDuration);
    }
}
```

This call reverts if the maximum number of reward tokens has already been added to the ibgtVault.

```
function addReward(address _rewardsToken, uint256 _rewardsDuration)
    external
    onlyInfrared
{
    if (_rewardsToken == address(0)) revert Errors.ZeroAddress();
    if (_rewardsDuration == 0) revert Errors.ZeroAmount();
>>  if (rewardTokens.length == MAX_NUM_REWARD_TOKENS) {
        revert Errors.MaxNumberOfRewards();
    }
    _addReward(_rewardsToken, infrared, _rewardsDuration);
}
```

Impact Explanation: High, as POL bribes cannot be auctioned as `BribesCollector::claimFees()` always reverts.

Likelihood Explanation: Low, as reward tokens can only be added by the infrared governance. For this issue to occur, the maximum number of reward tokens must have already been added.

Proof of Concept (if required): Add the following test to `BribeCollector.t.sol`:

```
    function testClaimFeeRevertsMaxRewards() public {
        uint256 rewardsDuration = 7 days;
        vm.startPrank(address(infrared));
        for (uint160 i = 0; i < 9; i++) {
            ibgtVault.addReward(address(i + 900), rewardsDuration);
        }
        vm.stopPrank();

        // set collectBribesWeight 50%
        vm.prank(infraredGovernance);
        infrared.updateInfraredBERABribesWeight(1e6 / 2);

        address searcher = address(777);

        // Arrange
        address recipient = address(3);
        address[] memory feeTokens = new address[](2);
        feeTokens[0] = address(wbera);
        feeTokens[1] = address(honey);

        uint256[] memory feeAmounts = new uint256[](2);
        feeAmounts[0] = 1 ether;
        feeAmounts[1] = 2 ether;

        // simulate bribes collected by the collector contract
        deal(address(wbera), address(collector), 1 ether);
        deal(address(honey), address(collector), 2 ether);

        address payoutToken = collector.payoutToken();
        uint256 payoutAmount = collector.payoutAmount();

        // searcher approves payoutAmount to the collector contract
        // deal(payoutToken, searcher, payoutAmount);
        // since payoutToken is wbera, deal and deposit
        vm.deal(searcher, payoutAmount);
        vm.prank(searcher);
        wbera.deposit{value: payoutAmount}();

        // Act
        vm.startPrank(searcher);
        ERC20(payoutToken).approve(address(collector), payoutAmount);
        vm.expectRevert(abi.encodeWithSignature("MaxNumberOfRewards()"));
        collector.claimFees(recipient, feeTokens, feeAmounts);
        vm.stopPrank();
    }
```

Recommendation (optional): Consider adding wbera as a reward token for the ibgtVault when `Infrared::setIBGT()` is called.

3.2.9 Missing call to `harvestBoostRewards()` in `onReward()` for the ibgtVault

Severity: Medium Risk

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

Summary: When ibgtVault stakers claim their rewards, `onReward()` calls `harvestVault()` but not `harvestBoostRewards()` which means that the ibgtVault will not be notified of new rewards and the `rewardRate` is not updated.

Finding Description: Vault stakers can claim their rewards by calling `getReward()`. In the `onReward()` hook, `harvestVault()` is called to harvest any BGT rewards from the underlying `BerachainRewardsVault`. However, `harvestBoostRewards()` should also be called if the vault is the ibgtVault, since at the end of that function's call flow, the ibgtVault is notified of new rewards.

`Infrared::harvestBoostRewards()` calls `RewardsLib::harvestBoostRewards()`. This function gets the rewards from the `_bgtStaker` and calls `RewardsLib::_handleTokenRewardsForVault()` to notify the ibgtVault of the new rewards.

```
function _handleTokenRewardsForVault(
    RewardsStorage storage $,
    IInfraredVault _vault,
    address _token,
    address voter,
    uint256 _amount,
    uint256 _feeTotal,
    uint256 _feeProtocol,
    uint256 rewardsDuration
) internal {
    ... ...
    uint256 _amtVoter;
    uint256 _amtProtocol;

    // calculate and distribute fees on rewards
    (_amount, _amtVoter, _amtProtocol) =
        _chargedFeesOnRewards(_amount, _feeTotal, _feeProtocol);
    _distributeFeesOnRewards(
        $.protocolFeeAmounts, voter, _token, _amtVoter, _amtProtocol
    );

    // increase allowance then notify vault of new rewards
    if (_amount > 0) {
        ERC20(_token).safeApprove(address(_vault), _amount);
>>      _vault.notifyRewardAmount(_token, _amount);
    }
}
```

Impact Explanation: Medium, as vault stakers will lose out on rewards from boosts that should be notified to the vault.

Likelihood Explanation: Medium, as `Infrared::harvestBoostRewards()` can be called separately. However, this not being called in `onReward()` still causes loss for the stakers.

Proof of Concept (if required): Add the following test to `InfraredVault.t.sol`:

```
    function testHarvestBoostRewardsInVault() public {
        // Staking by User in InfraredBGT Vault
        uint256 stakeAmount = 1000 * 1e18;
        vm.startPrank(address(infrared));
        ibgt.mint(user, stakeAmount);
        vm.stopPrank();
        vm.startPrank(user);
        ibgt.approve(address(ibgtVault), stakeAmount);
        ibgtVault.stake(stakeAmount);
        vm.stopPrank();

        // Setup: Add reward token and distribute rewards
        MockERC20 rewardToken = new MockERC20("Reward", "RWD", 18);

        vm.startPrank(infraredGovernance);
        infrared.updateWhiteListedRewardTokens(address(rewardToken), true);
        infrared.addReward(address(ibgt), address(rewardToken), 7 days);
        vm.stopPrank();

        // add reward token rewards to vault
        deal(address(rewardToken), address(infrared), 10 ether);
        vm.startPrank(address(infrared));
        rewardToken.approve(address(ibgtVault), 10 ether);
        ibgtVault.notifyRewardAmount(address(rewardToken), 10 ether);
        vm.stopPrank();

        // Time Skip for Reward Distribution
        vm.warp(block.timestamp + 3 days); // Simulate passage of time for reward distribution

        vm.startPrank(user);
        uint256 rewardBalanceBERABefore = rewardToken.balanceOf(user);
        ibgtVault.getReward();
        uint256 rewardBalanceBERABetween = rewardToken.balanceOf(user);
        vm.stopPrank();

        uint256 rewardBalanceBERADifferenceBetween = rewardBalanceBERABetween - rewardBalanceBERABefore;
        assertEq(rewardBalanceBERADifferenceBetween, 4285714285714147000);

        vm.warp(block.timestamp + 4 days); // Simulate passage of time for reward distribution
        // Claiming Rewards and Checking Balances
        vm.startPrank(user);
        ibgtVault.getReward();
        uint256 rewardBalanceBERAAfter = rewardToken.balanceOf(user);
        vm.stopPrank();

        uint256 rewardBalanceBERADifferenceAfter = rewardBalanceBERAAfter - rewardBalanceBERABetween;
        uint256 rewardBalanceBERADifference = rewardBalanceBERABetween + rewardBalanceBERADifferenceAfter;
        assertEq(rewardBalanceBERADifference, 9999999999999676000);

        // add reward token rewards to vault
        deal(address(rewardToken), address(infrared), 10 ether);
        vm.startPrank(address(infrared));
        rewardToken.approve(address(ibgtVault), 10 ether);
        ibgtVault.notifyRewardAmount(address(rewardToken), 10 ether);
        vm.stopPrank();

        (,, uint256 periodFinishBefore, uint256 rewardRateBefore,,,) =
            ibgtVault.rewardData(address(rewardToken));

        assertEq(periodFinishBefore, block.timestamp + 7 days);

        // Time Skip for Reward Distribution
        vm.warp(block.timestamp + 3 days); // Simulate passage of time for reward distribution

        vm.startPrank(user);
        uint256 rewardBalanceBERABoostBefore = rewardToken.balanceOf(user);
        ibgtVault.getReward();
        uint256 rewardBalanceBERABoostBetween = rewardToken.balanceOf(user);
        vm.stopPrank();

        uint256 rewardBalanceBERABoostDifferenceBetween = rewardBalanceBERABoostBetween -
        ↪   rewardBalanceBERABoostBefore;
        assertEq(rewardBalanceBERABoostDifferenceBetween, 4285714285714406000);

        // this simulates harvestBoostRewards() being called and notifies the vault of 10 ether more reward
        ↪   tokens
        deal(address(rewardToken), address(infrared), 10 ether);
        vm.startPrank(address(infrared));
        rewardToken.approve(address(ibgtVault), 10 ether);
        ibgtVault.notifyRewardAmount(address(rewardToken), 10 ether);
        vm.stopPrank();

        (,, uint256 periodFinishAfter, uint256 rewardRateAfter,,,) =
            ibgtVault.rewardData(address(rewardToken));

        assertEq(periodFinishAfter, block.timestamp + 7 days);

        // reward rate increased
        assertTrue(rewardRateAfter > rewardRateBefore);

        vm.warp(block.timestamp + 4 days); // Simulate passage of time for reward distribution

        // Claiming Rewards and Checking Balances
        vm.startPrank(user);
        ibgtVault.getReward();
        uint256 rewardBalanceBERABoostAfter = rewardToken.balanceOf(user);
        vm.stopPrank();

        uint256 rewardBalanceBERABoostDifferenceAfter = rewardBalanceBERABoostAfter -
        ↪   rewardBalanceBERABoostBetween;
        uint256 rewardBalanceBERABoostDifference = rewardBalanceBERABoostBetween +
        ↪   rewardBalanceBERABoostDifferenceAfter;
        assertEq(rewardBalanceBERABoostDifference, 23265306122448584000);
        assertTrue(rewardBalanceBERABoostDifference > rewardBalanceBERADifference);
    }
```

Recommendation (optional): Consider calling `harvestBoostRewards()` in `onReward()` if the vault is the ibgtVault.

3.2.10 Unaccounted Staking Token as Rewards, Causing Permanent Fund Lock

Severity: Medium Risk

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

Summary: The `WrappedVault` contract does not account for staking token rewards received via `claimRewards` in the `totalAssets` calculation. This results in an inaccurate representation of the vault's assets, leading to inaccessible funds and permanent locking of staking token rewards.

Finding Description:

- Found in src/core/WrappedVault.sol at Line 116.

```
106:    function claimRewards() external {
        ...
116: =>             if (_token == asset) continue;
            ...
130:    }
```

The `claimRewards` function deliberately skips transferring staking token rewards to the `rewardDistributor` when the reward token is the staking token. However, this increased balance is not accounted for in the `totalAssets` calculation.

- Found in src/core/WrappedVault.sol at Line 75.

```
74:    function totalAssets() public view virtual override returns (uint256) {
75: =>         return iVault.balanceOf(address(this)) + deadShares;
76:    }
```

The `totalAssets` function is expected to reflect the total assets managed by the `WrappedVault`. However, it omits staking token rewards received via `claimRewards`, leading to a discrepancy between the actual balance and the reported `totalAssets`.

This breaks the security guarantee of asset accessibility and accurate reporting. Assuming the rewards in staking tokens are meant to be incentivized users' deposits, the omission of staking token rewards from `totalAssets` might:

1. Prevents users from withdrawing or redeeming these rewards.
2. Permanently locks the funds in the `WrappedVault`, rendering them inaccessible.

If the rewards are reserved for other purposes, the lacking of a withdrawal mechanism also prevents the funds from being accessed. Both cases result in permanently stuck funds (accrued rewards in terms of staking tokens).

Impact Explanation: The issue locks staking token rewards within the vault, making them permanently inaccessible. Users cannot redeem these rewards, and even administrators cannot withdraw them once all shares are burned. This directly impacts user incentives and causes a loss of funds, warranting a **High** severity rating.

Likelihood Explanation: The likelihood is **Medium** because:

1. The issue occurs only if the staking token is also set as the reward token.
2. The reward distribution mechanism must be configured in this specific manner, which may not be the default or common setup. However, the functionality exists, and the bug is systemic when the conditions are met.

Proof of Concept: Here is the relevant portion of the test case that demonstrates the issue:

```
function testClaimedRewardsInStakingTokenAreLoss() public {
    uint256 rewardsAmount = 100 ether;
    uint256 rewardsDuration = 30 days;

    // 1. Setup stakingToken as reward token for WrappedVault
    vm.startPrank(address(infrared));
    deal(address(stakingToken), address(infrared), rewardsAmount);
    stakingToken.approve(address(wrappedVault.iVault()), rewardsAmount);
    wrappedVault.iVault().notifyRewardAmount(address(stakingToken), rewardsAmount);
    vm.stopPrank();

    // 2. User deposits staking token
    vm.startPrank(user);
    stakingToken.approve(address(wrappedVault), 500 ether);
    wrappedVault.deposit(500 ether, user);
    skip(rewardsDuration + 100 minutes);

    // 3. Claim rewards after rewards duration has passed
    wrappedVault.claimRewards();
    assertGt(
        stakingToken.balanceOf(address(wrappedVault)),
        99.99 ether,
        "wrappedVault should hold staking token rewards"
    );

    // 4. BUG: totalAssets does not reflect staking token rewards
    assertEq(
        wrappedVault.totalAssets(),
        500 ether, // @audit: should be > 599.99
        "totalAssets should reflect the staking token rewards"
    );

    // 5. User withdraws but does not receive rewards
    wrappedVault.withdraw(wrappedVault.maxWithdraw(user), user, user);
    assertEq(
        stakingToken.balanceOf(user),
        500 ether, // @audit: user should receive 599.99
        "User's staking token balance should be restored"
    );

    // 6. Funds remain locked permanently
    assertGt(
        stakingToken.balanceOf(address(wrappedVault)),
        99.99 ether,
        "wrappedVault should be emptied after all shares are burned"
    );
}
```

POC description:

1. **Setup Rewards**: The staking token is configured as the reward token for the `WrappedVault`. A reward amount of 100 tokens is allocated.
2. **User Deposit**: A user deposits 500 tokens into the `WrappedVault`.
3. **Claim Rewards**: After the rewards duration passes, the user claims rewards. The vault's balance increases by the reward amount (approximately 100 tokens).
4. **Bug Observation**: The `totalAssets` function does not reflect the increased balance from the staking token rewards. It only accounts for the initial deposit of 500 tokens, instead of the expected total (over 599.99 tokens).
5. **User Withdrawal**: The user withdraws all their shares but does not receive a portion of the accrued staking token rewards. They only recover their initial deposit of 500 tokens.
6. **Permanent Locking**: After all shares are burned, the staking token rewards remain locked in the vault. These funds cannot be accessed or withdrawn, leading to permanent loss.

Recommendation: To fix the issue,.

1/ update the `totalAssets` function to include the balance of the staking token held by the contract. For example:

```
return iVault.balanceOf(address(this)) + deadShares + asset.balanceOf(address(this));
```

or 2/ implement an authorized `withdraw` function to assess the funds.

3.2.11 Missing call to `harvestVault()` in `stake()` and `withdraw()` leads to missed rewards

Severity: Medium Risk

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

Summary: When users stake or withdraw, `harvestVault()` is not called which means that the vault is not notified of all the new BGT rewards associated with the given staking token. This ultimately leads to stakers missing out on rewards.

Finding Description: Staked users can claim their rewards via `getReward()` or `getRewardForUser()`. Both functions call `harvestVault()` in the `onReward()` hook to notify the vault of new rewards. However, the vault isn't harvested when users stake or withdraw which leads to missed rewards. If users stake or withdraw during a reward period, the vault should be harvested so that it is notifed of the new rewards and the `rewardRate` and `periodFinish` are adjusted accordingly.

Impact Explanation: Medium, because stakers will miss out on rewards from vault harvesting.

Likelihood Explanation: Medium, because the vault isn't harvested every time users stake or withdraw. However, `harvestVault()` can be called separately.

Proof of Concept (if required): The following test needs to be added to `InfraredVault.t.sol`:

```
    function testStakeNoHarvestVault() public {
        address alice = address(0x02);
        address bob = address(0x03);
        uint256 stakeAmount = 100 ether;
        deal(alice, stakeAmount);
        deal(bob, stakeAmount * 2);

        // alice stakes
        vm.startPrank(alice);
        wbera.deposit{value: stakeAmount}();

        // User approves the infraredVault to spend their tokens
        wbera.approve(address(infraredVault), stakeAmount);

        // User stakes tokens into the infraredVault
        infraredVault.stake(stakeAmount);

        // Check user's balance in the infraredVault
        uint256 userBalanceAlice = infraredVault.balanceOf(alice);
        assertEq(userBalanceAlice, stakeAmount, "User balance should be updated");

        // Check total supply in the infraredVault
        uint256 totalSupply = infraredVault.totalSupply() - 1; // infared holds a balance of 1 wei in every
        ↪  vault
        assertEq(totalSupply, stakeAmount, "Total supply should be updated");

        // Check staking token transferred to berachain rewards infraredVault
        assertEq(
            wbera.balanceOf(address(infraredVault.rewardsVault())), stakeAmount
        );
        assertEq(wbera.balanceOf(address(infraredVault)), 0);
        assertEq(wbera.balanceOf(alice), 0);

        vm.stopPrank();

        // bob stakes
        vm.startPrank(bob);
        wbera.deposit{value: stakeAmount}();

        // User approves the infraredVault to spend their tokens
        wbera.approve(address(infraredVault), stakeAmount);

        // User stakes tokens into the infraredVault
        infraredVault.stake(stakeAmount);

        // Check user's balance in the infraredVault
        uint256 userBalanceBob = infraredVault.balanceOf(bob);
        assertEq(userBalanceBob, stakeAmount, "User balance should be updated");

        // Check total supply in the infraredVault
        totalSupply = infraredVault.totalSupply() - 1; // infared holds a balance of 1 wei in every vault
        assertEq(totalSupply, stakeAmount * 2, "Total supply should be updated");

        // Check staking token transferred to berachain rewards infraredVault
        assertEq(
            wbera.balanceOf(address(infraredVault.rewardsVault())), stakeAmount * 2
        );
        assertEq(wbera.balanceOf(address(infraredVault)), 0);
        assertEq(wbera.balanceOf(bob), 0);

        vm.stopPrank();
        // Setup reward token in the infraredVault and mint rewards
        vm.startPrank(address(infrared));
        infraredVault.updateRewardsDuration(address(rewardsToken), 604800);
        uint256 rewardsAmount = 100 ether;
        infraredVault.addReward(address(0x3), 604800);
        rewardsToken.mint(address(infrared), rewardsAmount);
        rewardsToken.approve(address(infraredVault), rewardsAmount);
        infraredVault.notifyRewardAmount(address(rewardsToken), rewardsAmount);
        vm.stopPrank();

        // check cannot recover reward token
        (,, uint256 periodFinishBefore, uint256 rewardRateBefore, uint256 lastUpdateTimeBefore,,) =
            infraredVault.rewardData(address(rewardsToken));
        assertEq(lastUpdateTimeBefore, block.timestamp);
        assertEq(periodFinishBefore, block.timestamp + 604800);
        assertTrue(rewardRateBefore > 0);

        vm.warp(block.timestamp + 3 days);

        // bob stakes again
        vm.startPrank(bob);
        wbera.deposit{value: stakeAmount}();

        // User approves the infraredVault to spend their tokens
        wbera.approve(address(infraredVault), stakeAmount);

        // User stakes tokens into the infraredVault
        infraredVault.stake(stakeAmount);

        // Check user's balance in the infraredVault
        uint256 secondUserBalanceBob = infraredVault.balanceOf(bob);
        assertEq(secondUserBalanceBob, stakeAmount * 2, "User balance should be updated");

        // Check total supply in the infraredVault
        totalSupply = infraredVault.totalSupply() - 1; // infared holds a balance of 1 wei in every vault
        assertEq(totalSupply, stakeAmount * 3, "Total supply should be updated");

        // Check staking token transferred to berachain rewards infraredVault
        assertEq(
            wbera.balanceOf(address(infraredVault.rewardsVault())), stakeAmount * 3
        );
        assertEq(wbera.balanceOf(address(infraredVault)), 0);
        assertEq(wbera.balanceOf(bob), 0);

        vm.stopPrank();

        // but harvestVault() is not called which means that the rewardRate etc. stays the same
        (,, uint256 periodFinishAfter, uint256 rewardRateAfter, uint256 lastUpdateTimeAfter,,) =
            infraredVault.rewardData(address(rewardsToken));
        assertEq(lastUpdateTimeAfter, lastUpdateTimeBefore + 3 days);
        assertEq(periodFinishAfter, periodFinishBefore);
        assertEq(rewardRateAfter, rewardRateBefore);
    }
```

Recommendation (optional): Consider harvesting the vault when users stake and withdraw.

3.2.12 Missing Fee Application on InfraredBERA Portion of Collected Bribes

Severity: Medium Risk

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

Summary: in `RewardsLib::harvestOperatorRewards()` fees are only applied to `amtIbgtVault` portion of bribes, excluding `amtInfraredBERA` portion from fee charges, causing loss of funds to the voter and protocol.

Finding Description: The call flow shows fees are only charged on the IBGT vault portion:

```
// BribeCollector.sol
function claimFees(...) {
    // Transfers payout token and calls Infrared
    infrared.collectBribes(payoutToken, payoutAmount);
}

// Infrared.sol
function collectBribes(address _token, uint256 _amount) {
    _rewardsStorage().collectBribesInWBERA(
        _amount,
        address(wbera),
        address(ibera),
        address(ibgtVault),
        address(voter),
        rewardsDuration()
    );
}

// RewardsLib.sol
function collectBribesInWBERA(...) {
    // Split amount
    amtInfraredBERA = _amount * $.collectBribesWeight / WEIGHT_UNIT;
    amtIbgtVault = _amount - amtInfraredBERA;

    // Fees only applied to amtIbgtVault
    _handleTokenRewardsForVault(
        $,
        IInfraredVault(ibgtVault),
        wbera,
        voter,
        amtIbgtVault,  // <-- Only this portion gets fees
        feeTotal,
        feeProtocol,
        rewardsDuration
    );
}
```

And during `RewardsLib::harvestOperatorRewards()`, Fees are applied to the portion of the `shareholder-Fees` of `amtInfraredBERA` that was initially sent, causing loss of fees that should have been applied to the whole `amtInfraredBERA`.

This also happens during `infrared::harvestBase()` flow.

Impact Explanation:

- Loss of fee revenue on `amtInfraredBERA` portion.

Likelihood Explanation: High likelihood as this occurs on every bribe collection transaction.

Recommendation: Apply fees to total bribe amount before splitting:

3.2.13 `addIncentives` check for `isWhitelisted` rewardTokens is not effective

Severity: Medium Risk

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

Vulnerability Discussion: When adding Incentives for a reward token in a vault a check is made to see if the token is whitelisted.. This check is done by checking the reward duration if there is a duration set the token is assumed to be whitelisted, This check is not effective as the token might have been removed and duration is never reset on removal.. .

```
    function addIncentives(//@audit-issue the check for is whitelisted is flawed
        address _stakingToken,
        address _rewardsToken,
        uint256 _amount
    ) external {
        _vaultStorage().addIncentives(_stakingToken, _rewardsToken, _amount);
    }
```

.

```
    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) {//@audit-issue the check for is whitelisted is flawed
            revert Errors.RewardTokenNotWhitelisted();
        }

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

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

POC: We first add the reward token after whitelist then removed from whitelist the check to see if the token is whitelisted does noting.

```
        function test_Flawed_Whitelist_Check() public {
        // Setup new reward token
        uint256 rewardsDuration = 7 days;
        deal(address(ibera), user1, 2000 ether);


        vm.startPrank(user1);
        ibera.approve(address(infraredDistributor), type(uint256).max);
        vm.stopPrank();


        vm.startPrank(infraredGovernance);
        // Whitelist the new reward token first
        infrared.updateWhiteListedRewardTokens(address(ibera), true);

        // governance adds the reward tokens
         infrared.addReward(
            address(wbera), address(ibera), rewardsDuration //set to 7 days
        );

        //governance adds a validator
        ValidatorTypes.Validator memory validator_str = ValidatorTypes.Validator({
            pubkey: "0x1234567890abcdef",
            addr: address(validator)
        });

        ValidatorTypes.Validator[] memory validators =
            new ValidatorTypes.Validator[](1);
        validators[0] = validator_str;
        bytes[] memory pubkeys = new bytes[](1);
        pubkeys[0] = validator_str.pubkey;
        infrared.addValidators(validators);

        //Remove from  Whitelist
        infrared.updateWhiteListedRewardTokens(address(ibera), false);
        vm.stopPrank();

        vm.prank(user1);
        infrared.addIncentives(address(wbera),  address(ibera), 100 ether);

        vm.startPrank(validator);
        infraredDistributor.claim( "0x1234567890abcdef",  user1);

        vm.stopPrank();

    }
```

. **Impact:** users can keep adding incentives for non whitelisted tokens. . **Recommendation:** check if the token is whitelisted through the `whitelistedRewardTokens` function.

3.2.14 IBGT minting may block all reward harvesting operations

Severity: Medium Risk

Context: RewardsLib.sol#L154-L155

Summary: Harvest operations will revert entirely if the IBGT token is paused since there's no error handling around IBGT minting.

Finding Description:

`harvestVault()` attempts to mint IBGT when harvesting BGT rewards:

```
// Mint InfraredBGT tokens equivalent to the BGT rewards
IInfraredBGT(ibgt).mint(address(this), bgtAmt); // @audit-poc what if it is paused?
```

Since IBGT inherits from ERC20PresetMinterPauser, minting can be disabled via pause(). Unlike RED minting which has try-catch error handling, IBGT minting will revert the entire transaction if paused.

High impact, because a paused IBGT token will prevent:

- All reward harvesting operations.
- Distribution of BGT rewards to users.
- Core protocol functionality around liquid staking.
- The entire reward claiming process.

Likelihood is low, because:

- IBGT would only be paused in emergency scenarios.
- Pause functionality is controlled by designated protocol governors.
- PAUSER_ROLE has strict access controls.

Proof of concept:

```
function testHarvestRevertOnIBGTPause() public {
    // Setup initial state
    vm.startPrank(infraredGovernance);
    infrared.updateWhiteListedRewardTokens(address(wbera), true);
    vm.stopPrank();

    // Setup user stake
    address user = address(123);
    vm.deal(user, 100 ether);
    vm.startPrank(user);
    wbera.deposit{value: 100 ether}();
    wbera.approve(address(infraredVault), 100 ether);
    infraredVault.stake(100 ether);
    vm.stopPrank();

    // Setup vault operator
    vm.startPrank(address(infraredVault));
    infraredVault.rewardsVault().setOperator(address(infrared));
    vm.stopPrank();

    // Setup rewards
    address vaultWbera = factory.getVault(address(wbera));
    vm.startPrank(address(blockRewardController));
    bgt.mint(address(distributor), 10 ether);
    vm.stopPrank();

    vm.startPrank(address(distributor));
    bgt.approve(address(vaultWbera), 10 ether);
    IBerachainRewardsVault(vaultWbera).notifyRewardAmount(
        abi.encodePacked(bytes32("v0"), bytes16("")),
        10 ether
    );
    vm.stopPrank();
    // Let rewards accrue
    vm.warp(block.timestamp + 7 days);

    // Pause IBGT
    vm.prank(infraredGovernance);
    InfraredBGT(address(ibgt)).pause();

    // Harvest should revert due to IBGT being paused
    vm.prank(keeper);
    vm.expectRevert();
    infrared.harvestVault(address(wbera));
}
```

Recommendation: Add try-catch error handling around IBGT minting similar to RED token handling.

This allows harvest operations to fail gracefully if IBGT is paused rather than reverting entirely.

3.2.15 Calling `InfraredBERAWithdraworLite::sweep()` breaks the accounting of deposits and results in users receiving less shares

Severity: Medium Risk

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

Summary: Whenever `InfraredBERAWithdraworLite::sweep()` is called to re-stake principal of forced withdrawals, a fee is deducted. Ultimately, this results in users receiving less shares than they should.

Finding Description: When `InfraredBERA::mint()` is called to mint iBERA shares to the receiver for BERA paid in by the sender, `InfraredBERA::_deposit()` is called which queues the amount of BERA that the sender paid minus the fees in the `InfraredBERADepositor`. The `deposits` of BERA backing InfraredBERA is increased by that amount.

```
function _deposit(uint256 value)
    private
    returns (uint256 nonce, uint256 amount, uint256 fee)
{
    // @dev check at internal deposit level to prevent donations prior
    if (!_initialized) revert Errors.NotInitialized();

    // calculate amount as value less deposit fee
    uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT;
    fee = InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
    if (value < min + fee) revert Errors.InvalidAmount();

    amount = value - fee;
    // update tracked deposits with validators
>>  deposits += amount;
    // escrow funds to depositor contract to eventually forward to precompile
    nonce = IInfraredBERADepositor(depositor).queue{value: value}(amount);
}
```

`InfraredBERADepositor::reserves()` now includes that amount. The queued deposit can be executed by calling `InfraredBERADepositor::execute()`. If that happens, the funds will be deposited in a validator. If that validator is force exited, the funds can be retrieved when the keeper calls `InfraredBERAWithdrawor-Lite::sweep()`. This function queues the staked amount again.

```
function sweep(bytes calldata pubkey) external {
    // only callable when withdrawals are not enabled
    if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
        revert Errors.Unauthorized(msg.sender);
    }
    // onlyKeeper call
    if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
        revert Errors.Unauthorized(msg.sender);
    }
    // Check if validator has already exited - do this before checking stake
    if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
        revert Errors.ValidatorForceExited();
    }
    // forced exit always withdraw entire stake of validator
    uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

    // do nothing if InfraredBERA deposit would revert
    uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
        + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
    if (amount < min) return;
    // revert if insufficient balance
    if (amount > address(this).balance) revert Errors.InvalidAmount();

    // todo: verfiy forced withdrawal against beacon roots

    // register new validator delta
    IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

    // re-stake amount back to ibera depositor
>>  IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
        value: amount
    }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

    emit Sweep(InfraredBERA, amount);
}
```

However, the `MINIMUM_DEPOSIT_FEE` is again deducted from the amount that is queued. This means that the amount returned by `reserves()` will be 1 ether less than what it was before the amount was queued the first time but the amount stored in `deposits` is still the same.

If a user minted for 10002 ether, the amount stored in `deposits` would be 10001 since a fee of 1 ether is charged. `reserves()` is also increased by 10001 ether. When `InfraredBERADepositor::execute()` is called for that amount and later swept, the `deposits` will still be 10001 ether, but `reserves()` will be 10000 ether because the fee is charged again.

Impact Explanation: Medium, as the `deposits` amount will be too high because 1 ether flows out of the system but `deposits` remains unchanged. This results in the accounting of the deposits being broken and the iBERA <=> BERA exchange rate being wrong. As a result of that, new shares are minted with the wrong `deposits` amount and users receive less shares than they should.

If the `WithdraworLite` is eventually upgraded to the `Withdrawor`, the broken accounting makes it so that not all users can withdraw all their shares which may lead to a bankrun scenario. If needed, a PoC can also be provided for this.

Likelihood Explanation: High, as `deposits` is unchanged and the accounting further breaks every time `InfraredBERAWithdraworLite::sweep()` is called. Since forced exits are a core functionality, the issue is certain to occur.

Proof of Concept (if required): Add the following test to `InfraredBERADepositor.t.sol`:

It is important to note that even though this test calls the sweep function of the `InfraredBERAWithdrawor.sol`, this function is identical to `InfraredBERAWithdraworLite::sweep()`. It was easier to use the withdrawor in the current test setup but this does not change the validity of the issue.

```
    function testSweepStakesWrongAccounting() public {
        vm.deal(address(ibera), 10000 ether);
        uint256 deposits = ibera.deposits();
        uint256 totalSupply = ibera.totalSupply();
        uint256 sharesIbera = ibera.balanceOf(address(ibera));
        assertEq(deposits, 10 ether);
        assertEq(totalSupply, 10 ether);
        assertEq(sharesIbera, 10 ether);
        vm.prank(address(ibera));
        (, uint256 shares_) = ibera.mint{value: 9991 ether}(address(ibera));

        assertEq(ibera.balanceOf(address(ibera)), sharesIbera + shares_);
        assertEq(ibera.totalSupply(), totalSupply + shares_);
        assertEq(shares_, 9990 ether);

        assertEq(ibera.deposits(), deposits + 9990 ether);
        assertEq(ibera.totalSupply(), totalSupply + 9990 ether);
        assertEq(ibera.balanceOf(address(ibera)), 10000 ether);
        assertEq(depositor.reserves(), 10000 ether);

        vm.prank(infraredGovernance);
        ibera.setDepositSignature(pubkey0, signature0);

        vm.startPrank(keeper);
        depositor.execute(pubkey0, 10000 ether);
        vm.stopPrank();

        // Get current stake from setup
        uint256 validatorStake = ibera.stakes(pubkey0);
        assertEq(validatorStake, 10000 ether);

        // Disable withdrawals (required for sweep)
        vm.prank(infraredGovernance);
        ibera.setWithdrawalsEnabled(false);

        // Simulate forced withdrawal by dealing ETH to withdrawor
        vm.deal(address(withdrawor), validatorStake);

        // Test successful sweep
        vm.prank(keeper);
        withdrawor.sweep(pubkey0);

        // Verify stake and balance after sweep
        assertEq(ibera.stakes(pubkey0), 0, "Stake should be zero after sweep");
        assertEq(
            address(withdrawor).balance,
            0,
            "Withdrawor balance should be zero after sweep"
        );

        // reserves are only 9999 ether but deposits are 10000 ether
        // this means that shares are worth less than 1:1
        assertEq(depositor.reserves(), 9999 ether);
        assertEq(ibera.deposits(), 10000 ether);
        assertEq(ibera.totalSupply(), 10000 ether);

        // shares bob actually should get with 10001 ether amount: totalSupply * amount / deposits
        // 10000 ether * 10001 ether / 9999 ether;
        uint256 sharesBobActual = ibera.totalSupply() * 10001000000000000000000 / 9999000000000000000000;

        // bob comes in with 10002 ether, 10001 ether after shares
        vm.prank(bob);
        (, uint256 sharesBob_) = ibera.mint{value: 10002 ether}(bob);
        assertEq(sharesBob_, 10001 ether);
        uint256 depositsAfterBob = ibera.deposits();
        assertEq(depositor.reserves(), 20000 ether);
        assertEq(depositsAfterBob, 20001 ether);
        assertTrue(sharesBobActual > sharesBob_);
    }
```

Recommendation (optional): Consider reducing the amount of deposits when `sweep()` is called.`InfraredBERA` should have a function to decrease the amount stored in `deposits` when `InfraredBERAWithdraworLite::sweep()` is called. Alternatively, do not charge a fee for deposits that are queued due to `sweep()` being called..

3.2.16 Not calling harvestBase during interactions in InfraredBERA may lead to unfair distribution of rewards

Severity: Medium Risk

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

Summary: The `Infrared::harvestBase` function redeems excess BGT for BEAR and sends it to `IInfraredBERAReceivor` as a reward. These rewards are then deposited into `InfraredBERA` to increase the value of shares.. However, if `harvestBase` is not called during interactions between `InfraredBERA` and the mint/burn functions, it may lead to unfair reward distribution.

Finding Description: When the `InfraredBERA::mint` and `InfraredBERA::redeem` functions are called, the `compound` function is invoked to attempt to harvest rewards..

```
    /// @inheritdoc IInfraredBERA
    function mint(address receiver)
        public
        payable
        returns (uint256 nonce, uint256 shares)
    {
        // compound yield earned from EL rewards first
        compound();
        ...

    /// @inheritdoc IInfraredBERA
    function burn(address receiver, uint256 shares)
        external
        payable
        returns (uint256 nonce, uint256 amount)
    {
        if (!withdrawalsEnabled) revert Errors.WithdrawalsNotEnabled();
        // compound yield earned from EL rewards first
        compound();
        ...
```

If the rewards exceed `InfraredBERAConstants.MINIMUM_DEPOSIT`, then `InfraredBERAConstants.MINIMUM_DEPOSIT_FEE` will be added to `InfraredBERA`. These rewards come from `Infrared::harvestBase`..

```
    /// @inheritdoc IInfraredBERAFeeReceivor
    function sweep() external returns (uint256 amount, uint256 fees) {
        (amount, fees) = distribution();
        // do nothing if InfraredBERA deposit would revert
        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (amount < min) return (0, 0);

        // add to protocol fees and sweep amount back to ibera to deposit
        if (fees > 0) shareholderFees += fees;
        IInfraredBERA(InfraredBERA).sweep{value: amount}();
        emit Sweep(InfraredBERA, amount, fees);
    }
```

However, the `compound` function does not call `Infrared::harvestBase` to fetch any potential new rewards, which may result in rewards not being added to `InfraredBERA` in a timely manner, leading to unfair distribution.

Impact Explanation:

`InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE` is 11 ether, so the impact of adding this portion of the reward cannot be ignored. **Likelihood Explanation:** It is very likely that the following scenario could occur: after calling `harvestBase()`, the rewards reach `InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE`, but before the call, the rewards do not reach this threshold.

Proof of Concept:

Scenario 1:

1. `InfraredBERAFeeReceivor` has accumulated 10.5 ether, and after calling `Infrared::harvestBase`, it can generate an additional 0.5 ether in rewards.
2. If the user does not manually call `Infrared::harvestBase` and directly burns the shares to exit, they will miss out on the rewards they should have received.

Scenario 2:

1. `InfraredBERAFeeReceivor` has accumulated 10.5 ether, and after calling `Infrared::harvestBase`, it can generate an additional 0.5 ether in rewards.

2. User A can first mint shares and then call `harvestBase`, allowing them to participate in the distribution of this 11 ether in rewards.. However, if the user calls `harvestBase` first and then mints, they will not be able to receive the reward distribution for this round.

Recommendation: It is recommended to check in the `InfraredBERAFeeReceivor::sweep` function whether the rewards from calling `Infrared::harvestBase` can exceed the deposit limit. If so, `Infrared::harvestBase` should be called.

3.2.17 Rewards from the vault's rewardsVault might be receiving the wrong `redMintRate`

Severity: Medium Risk

Context: RewardsLib.sol#L481-L498

Vulnerability Details: In `RewardsLib.sol`'s `updateRedMintRate`:

```
function updateRedMintRate(RewardsStorage storage $, uint256 _iredMintRate)
    external
{
    // Update the RED minting rate
    // This rate determines how many RED tokens are minted per IBGT

    // @note The rate can be greater than RATE_UNIT (1e6)
    // This allows for minting multiple RED tokens per IBGT if desired

    // For example:
    // - If _iredMintRate = 500,000 (0.5 * RATE_UNIT), 0.5 RED is minted per IBGT
    // - If _iredMintRate = 2,000,000 (2 * RATE_UNIT), 2 RED are minted per IBGT

    // The actual calculation is done in harvestVault
    // uint256 _redAmt = Math.mulDiv(_bgtAmt, $.redMintRate, RATE_UNIT);

    $.redMintRate = _iredMintRate;
}
```

`$.redMintRate` is updated to the new mint rate. However, before the update, `harvestVault` should be called first as `harvestVault` uses `$.redMintRate` to decide the amount of RED tokens to mint in accordance to the rewards.

And current accumulated rewards should be receiving the old red mint rate since they were accumulated during the period where the old mint rate was in effect. And everyone who contributed might have contributed because they wanted that old mint rate.

Recommendation: Run `harvestVault` before redMintRate is updated so that current accumulated rewards are able to receive the old mint rate which is the rate that stakers are aware of when they staked.

3.2.18 When pausing RED, some vaults will lose on rewards and other won't on `harvestVault()`

Severity: Medium Risk

Context: RewardsLib.sol#L180

Summary: in `infrared::harvestVault()`, when RED token is paused, some vaults will lose on RED tokens while others won't, causing unfair reward distribution and loss of rewards (RED) for some users.

Finding Description: during `infrared::harvestVault()` for specific staking token, we wrap the RED token minting logic in a try/catch method to avoid pausing reward distribution logic hook on `infraredVault` when RED token gets paused.

the problem in this implementation is that if RED token gets paused, it retroactively affects previous rewards.

- Since rewards stake their tokens first, then they get credited `iBGT` during `rewardsLib::harvestVault()`.
- RED minted to that vault is a portion of the iBGT rewards as seen from the mintRate calculation.

```
File: RewardsLib.sol
174:         uint256 mintRate = $.redMintRate;

179:             uint256 redAmt = bgtAmt * mintRate / RATE_UNIT;
```

- Now there can be two `infraredVault` of wETH, wBTC staking for 1 week.
- The vault earned 100e18 BGT.
- RED mint rate is 100%.
- Admin submit a txn to pause RED token.
- a staker of infraredVault of wBTC front run it calling `harvestVault()` to not lose that RED mints to his vault.
- Admin txn pause RED after.
- The same user call `harvestVault()` to all other `infraredVaults()` to force them to lose on those RED minted proportionally to iBGT, he is incentivized to do so to not dilute the total supply of RED and make the RED sent to his own vault more valuable. **Impact Explanation:** Medium:
- Users lose rewards relative to other vaults in some conditions. **Likelihood Explanation:** Low:
- Pausing RED generally is a low probability event. **Recommendation (optional):** Either:

1. try to implement a script off chain to call `harvestVault()` to all current vaults.
2. implement a loop to call `harvestVault()` to all staking token inside RED pause Logic.
3. store an accounting of the not minted RED.

   NOTE!: Its worth noting that excluding mint() call of RED won't work, since we will need transfer call when calling `vault.notifyRewardAmount()` after the mint inside the try block.

3.2.19 `updateFee` in `RewardsLib.sol` should harvest first before updating fee so that current funds are charged with the correct fee

Severity: Medium Risk

Context: RewardsLib.sol#L447-L454

Vulnerability Details: This is updateFee in RewardsLib.sol:

```
function updateFee(
    RewardsStorage storage $,
    ConfigTypes.FeeType _t,
    uint256 _fee
) external {
    if (_fee > FEE_UNIT) revert Errors.InvalidFee();
    $.fees[uint256(_t)] = _fee;
}
```

- However these 3 functions: `harvestVault`, `harvestBoostRewards` and `harvestOperatorRewards` calculate the fees based on `$.fees[]`.
- Hence, by updating the fee without first harvesting the funds in vault, boost or operator rewards will cause current funds to be charged the wrong fee in the future..
- Since the funds were accumulated in the era of the current fee, they should be charged the current fee and not the new fee that is just about to be updated with.

Recommendation: Include 3 new lines where these 3 functions are called before the line `$.fees[uint256(_t)] = _fee;` is ran.

3.2.20 `harvestVault` might be permanently bricked if number of reward tokens in vault is already 10

Severity: Medium Risk

Context: RewardsLib.sol#L183-L187

Vulnerability Details: In `RewardsLib.sol`'s `harvestVault`, we can see that:

```
function harvestVault(
    RewardsStorage storage $,
    IInfraredVault vault,
    address bgt,
    address ibgt,
    address voter,
    address red,
    uint256 rewardsDuration
) external returns (uint256 bgtAmt) {
    ....
    ....
    if (red != address(0) && mintRate > 0) {
        // Calculate the amount of RED tokens to mint
        uint256 redAmt = bgtAmt * mintRate / RATE_UNIT;
        try IRED(red).mint(address(this), redAmt) {
            {
                // Check if RED is already a reward token in the vault
                (, uint256 redRewardsDuration,,,,,) = vault.rewardData(red);
                if (redRewardsDuration == 0) {
                    // Add RED as a reward token if not already added
-->                 vault.addReward(red, rewardsDuration);
                }
            }
            ....
        } catch {
            emit RedNotMinted(redAmt);
        }
    }
}
```

We can see that only `IRED(red).mint(address(this), redAmt)` can be caught by the `try`. `vault.addReward` however, will cause the whole function to revert if it fails.

Since vault's addReward limits the total reward tokens to 10:

```
if (rewardTokens.length == MAX_NUM_REWARD_TOKENS) {
    revert Errors.MaxNumberOfRewards();
}
```

If the vault reward token list length is already 10 then `harvestVault` will **always revert**

Impact: Vault cannot be harvested anymore.

Also, claiming rewards in `MultiRewards.sol` will be permanently bricked as it calls `onReward()` which is overriden to call `IInfrared(infrared).harvestVault(address(stakingToken))`. Hence, it will revert as well due to harvestVault reverting.

Recommendation: The `try` not only needs to catch error from mint but also the addReward. Or alternatively a simpler way would be to use an if statement to not mint RED if the length is already 10.

3.2.21 Clash between Infrared.sol's `updateInfraredBERABribesWeight` and BribeCollector's `claimFees` causes users to lose funds

Severity: Medium Risk

Context: BribeCollector.sol#L70-L98, RewardsLib.sol#L439-L445

Vulnerability Detail:

`claimFees` of BribeCollector.sol:

```
function claimFees(
    address _recipient,
    address[] calldata _feeTokens,
    uint256[] calldata _feeAmounts
) external {
    if (_feeTokens.length != _feeAmounts.length) {
        revert Errors.InvalidArrayLength();
    }
    if (_recipient == address(0)) revert Errors.ZeroAddress();
    // transfer price of claiming tokens (payoutAmount) from the sender to this contract
    ERC20(payoutToken).safeTransferFrom(
        msg.sender, address(this), payoutAmount
    );
    // increase the allowance of the payout token to the infrared contract to be send to
    // validator distribution contract
    ERC20(payoutToken).safeApprove(address(infrared), payoutAmount);
    // Callback into infrared post auction to split amount to vaults and protocol
    infrared.collectBribes(payoutToken, payoutAmount);
    // payoutAmount will be transferred out at this point

    // From all the specified fee tokens, transfer them to the recipient.
    for (uint256 i = 0; i < _feeTokens.length; i++) {
        address feeToken = _feeTokens[i];
        uint256 feeAmount = _feeAmounts[i];
        ERC20(feeToken).safeTransfer(_recipient, feeAmount);
        emit FeesClaimed(msg.sender, _recipient, feeToken, feeAmount);
    }
}
```

The above calls Infrared.sol's `collectBribes` which calls `collectBribesInWBERA` in RewardsLib.sol:

```
function collectBribesInWBERA(
    RewardsStorage storage $,
    uint256 _amount,
    address wbera,
    address ibera,
    address ibgtVault,
    address voter,
    uint256 rewardsDuration
) external returns (uint256 amtInfraredBERA, uint256 amtIbgtVault) {
    if (ibera == address(0)) revert Errors.ZeroAddress();
    ERC20(wbera).safeTransferFrom(msg.sender, address(this), _amount);

    // determine proportion of bribe amount designated for InfraredBERA
    amtInfraredBERA = _amount * $.collectBribesWeight / WEIGHT_UNIT;
    amtIbgtVault = _amount - amtInfraredBERA;

    address rec = IInfraredBERA(ibera).receivor();
    if (rec == address(0)) revert Errors.ZeroAddress();
    // Redeem WBERA for BERA and send to IBERA receivor
    IWBERA(wbera).withdraw(amtInfraredBERA);
    SafeTransferLib.safeTransferETH(rec, amtInfraredBERA);

    // get total and protocol fee rates
    uint256 feeTotal =
        $.fees[uint256(ConfigTypes.FeeType.HarvestBribesFeeRate)];
    uint256 feeProtocol =
        $.fees[uint256(ConfigTypes.FeeType.HarvestBribesProtocolRate)];

    _handleTokenRewardsForVault(
        $,
        IInfraredVault(ibgtVault),
        wbera,
        voter,
        amtIbgtVault,
        feeTotal,
        feeProtocol,
        rewardsDuration
    );
}
```

We can observe that `$.collectBribesWeight` decides the ratio between what is sent to the fee receivor vs **the vault**

Explanation: When the user decides to call `claimFees` of BribeCollector.sol here is the tradeoff.

What the user pays:

1. `payoutAmount` worth of `payoutToken`.

What the user can potential benefit from.

1. User can get all fee tokens in balance of BribeCollector.sol.
2. User might be a large staker in vault and hence would also benefit significantly from the % of `payoutAmount` sent to vault (which would go to rewarding stakers like the user as `vault.notifyRewardAmount` is called).

Hence when the user is weighing whether to do the bribe, the % of amount getting sent over to the vault is one of the concerns taken into calculations especially if the user is a big staker in the vault.

However, the variable deciding the % amount sent over is `$.collectBribesWeight` which is dynamic and can change if `updateInfraredBERABribesWeight` is called.

Impact: Hence, the user might end up making an unoptimal decision if transactions `updateInfraredBERABribesWeight` and `claimFees` are made around the same time and `updateInfraredBERABribesWeight` gets processed first by the blockchain.

- Because it would be the old weight when the user is sending the transaction so it is not possible for them to be aware that they will be losing money by doing this bribe trade.

Recommendation: It would be good to let the user pass in a minimum weight (kinda like slippage protection) and then require that `$.collectBribesWeight` is atleast that minimum.

3.2.22 Users may not get their full rewards when using the exit function

Severity: Medium Risk

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

Finding Description: When users deposit into the InfratedVault, users rewards are calculated with the updateReward modifier. When the vault accumulates rewards, the Infared.harvestVault() function needs to be called to send rewards to the InfraredVault. When users wants to get their rewards or want to exit the vault, the onReward() in the getRewardForUser() function calls the harvestVault function to get the rewards for the vault..

```
function getRewardForUser(address _user)
        public
        nonReentrant
        updateReward(_user)
    {
        onReward();
        //......

    }
```

The problem is that, due to the updateReward modifier being called before the onReward() function, users' rewards can be outdated. Users who use the exit function to leave the vault will get fewer rewards than they should.

```
function exit() external {
        withdraw(_balances[msg.sender]);
        getReward();
    }
```

When the exit() function is called, it first withdraws the staking token to the user and then calls the getReward function. In the getReward function, the user's rewards are updated with updateReward, and after that, vault rewards are accumulated with onReward. Therefore, the user won't receive any rewards for the accumulated rewards from the onReward call. **Impact Explanation:** Loss of rewards.

Proof of Concept: InfraredVault.t.sol.

```
function testgetRewardWithAndWithoutHarvestVault() public {

        uint256 rewardsAmount = 100 ether;
        uint256 rewardsDuration = 30 days;
        setUpGetReward(rewardsAmount, rewardsDuration);


        InfraredVault vault = infraredVault;
        address vaultWbera = factory.getVault(address(wbera));

        vm.startPrank(address(blockRewardController));
        bgt.mint(address(distributor), 100 ether);
        vm.stopPrank();

        vm.startPrank(address(distributor));
        bgt.approve(address(vaultWbera), 100 ether);
        IBerachainRewardsVault(vaultWbera).notifyRewardAmount(
            abi.encodePacked(bytes32("v0"), bytes16("")), 100 ether
        );
        vm.stopPrank();

        // Step 4: Passage of Time for Rewards Distribution
        vm.warp(block.timestamp + 10 days); // Simulating 10 days for reward accrual

        // Step 5: Harvest Vault - Distributing Rewards

        //infrared.harvestVault(address(wbera));
        vm.startPrank(user);
        // Manipulate time to simulate the passage of the reward duration
        skip(rewardsDuration + 100 minutes);
        infraredVault.exit();

        vm.stopPrank();

        // Check user's rewards token balance
        uint256 userRewardsBalance = rewardsToken.balanceOf(user);
        console.log("User Rewards without the harvestVault()", userRewardsBalance);

    }
```

```
Logs:
  User Rewards without the harvestVault() 99999999999999964500
```

```
Logs:
  User Rewards with the harvestVault() 199999999999999929000
```

Recommendation: Users rewards needs to be updated after the onReward() call.

3.2.23 Governor is unable to "un-sign" for a pubkey

Severity: Medium Risk

Context: InfraredBERADepositor.sol#L136-L137

Vulnerability Details: In InfraredBERADepositor.sol's `execute` function:

```
function execute(bytes calldata pubkey, uint256 amount) external {
        ....
        ....
-->     bytes memory signature = IInfraredBERA(InfraredBERA).signatures(pubkey);
-->     if (signature.length == 0) revert Errors.InvalidSignature();
        ....
        ....
}
```

We can see that in `execute`, whether or not the pubkey is approved by the governor is checked with `IInfraredBERA(InfraredBERA).signatures(pubkey)`'s **length not being zero**

And in the signature function in InfraredBERA.sol:

```
function setDepositSignature(
    bytes calldata pubkey,
    bytes calldata signature
) external onlyGovernor {
    if (signature.length != 96) revert Errors.InvalidSignature();
    emit SetDepositSignature(
        pubkey, _signatures[keccak256(pubkey)], signature
    );
    _signatures[keccak256(pubkey)] = signature;
}
```

We can see that the parameter `signature` is forced to be length 96.

Impact: This means that if the governor wants to un-sign a pubkey that has been signed previously, they would have to set `_signatures[keccak256(pubkey)]` to 0. But `setDepositSignature` will cause it to revert.

- Un-signing for a pubkey is a senario that should be considered as validators can change pubkey which is also shown in the logic of `replaceValidator` in Infrared.sol where validars are allowed to change pubkeys.

Recommendation: Change `if (signature.length != 96) revert Errors.InvalidSignature();`. **To**: `if (signature.length != 96 && signature.length != 0) revert Errors.InvalidSignature();`.

3.2.24 Malicious validators can block purge operations

Severity: Medium Risk

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

Description: When the protocol wants to remove a validator from the validator set, it uses `Infrared.sol::removeValidators()`, which calls `InfraredDistributor.sol::remove()`.

The `remove` function sets the final reward accumulator of the validator's snapshot (`s.amountCumulativeFinal`) to `amountsCumulative`:

```
    /// @inheritdoc IInfraredDistributor
    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);
    }
```

After `remove`, if the validator has no unclaimed rewards, they can be fully removed from the state by calling `purge`.

The `purge` function checks whether the validator's last reward claim amount is equal to their final amount. If not, it means the validator has unclaimed rewards, and the transaction will revert:

```
    /// @inheritdoc IInfraredDistributor
    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);
    }
```

Problem: Since anybody can call `notifyRewardAmount`, a malicious validator who has claimed all their rewards could front-run the admin's call to `remove` by calling `notifyRewardAmount` with a very tiny amount (equal to the number of validators). This would leave 1 wei as a "dust" amount, preventing their data from being purged because `s.amountCumulativeLast` would differ by 1 wei from `s.amountCumulativeFinal`, causing the call to `purge` to revert.

Example Scenario:

- The protocol admin decides to remove a validator from the set and calls `removeValidators`.
- The malicious validator front-runs the admin's call by:
  - Claiming their rewards (if there is any).
  - Calling `notifyRewardAmount` with an amount equal to the number of validators (1 wei each).
- The admin's call to remove takes place:
- `s.amountCumulativeLast` == 10e18.
- `s.amountCumulativeFinal` == 10e18 + 1 wei.
- The admin calls `purge`, but the transaction reverts because the validator has not claimed their 1 wei reward.

Proof of Concept: The above scenario is implemented in the following PoC. To test it, add the test case below to `InfraredDistributor.t.sol`:

```
    function test_NinjaPreventPurge() public {
        vm.prank(address(infrared));
        distributor.add(pubkey1, validator1);
        infrared.addValidator(validator1);

        vm.prank(validator1);
        require(token.approve(address(distributor), 1 wei));

        vm.prank(user);
        distributor.notifyRewardAmount(10 ether);

        // malicious validator front-running to claim his rewards and then adding 1 wei to prevent the purge
        vm.prank(validator1);
        distributor.claim(pubkey1, validator1);

        vm.prank(validator1);
        distributor.notifyRewardAmount(1 wei);

        // admin removes the validator
        vm.prank(address(infrared));
        distributor.remove(pubkey1);

        // purge fails because of 1 wei remaining reward
        vm.prank(user);
        vm.expectRevert(Errors.ClaimableRewardsExist.selector);
        distributor.purge(pubkey1);
    }
```

Run the test:

```
$ forge t --mt test_NinjaPreventPurge
```

Recommendation: To fix this issue, restrict access to the `notifyRewardAmount`:

```diff
    /// @inheritdoc IInfraredDistributor
-   function notifyRewardAmount(uint256 amount) external {
+   function notifyRewardAmount(uint256 amount) external onlyInfrared {
        if (amount == 0) revert Errors.ZeroAmount();

        uint256 num = infrared.numInfraredValidators();
        if (num == 0) revert Errors.InvalidValidator();

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

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

3.2.25 BalanceOf call could brick WrappedVault.claimRewards

Severity: Medium Risk

Context: WrappedVault.sol#L117

Summary: If the `balanceOf` function reverts for any token in `WrappedVault.claimRewards` then the function itself will get bricked.

Finding Description:

`WrappedVault.claimRewards` iterates through all reward tokens and queries the balance before transferring them:

```
for (uint256 i; i < len; ++i) {
    ERC20 _token = ERC20(_tokens[i]);
    // Skip if the reward token is the staking token
    if (_token == asset) continue;
    uint256 bal = _token.balanceOf(address(this));
    if (bal == 0) continue;
    (bool success, bytes memory data) = address(_token).call(
        abi.encodeWithSelector(
            ERC20.transfer.selector, rewardDistributor, bal
        )
    );
```

If the `balanceOf` call reverts for any of them (e.g. malicious upgrade or faulty logic), then `claimRewards` can not be called anymore.

It is currently not possible to remove reward tokens or do a selective claim by passing the list of tokens to claim for.

Impact Explanation: High: loss of all rewards in the WrappedVault.

Likelihood Explanation: Low: reward tokens are added by governance.

Proof of Concept: On demand as per contest terms.

Recommendation: Enable governance to remove reward tokens or allow passing a list of tokens to claim.

3.2.26 InfraredBERAWithdraworLite's sweep checks are inaccurate

Severity: Medium Risk

Context: InfraredBERAWithdraworLite.sol#L135-L137

Vulnerability Details: In `InfraredBERAWithdraworLite.sol`'s `sweep()` function:

```
function sweep(bytes calldata pubkey) external {
    // only callable when withdrawals are not enabled
    if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
        revert Errors.Unauthorized(msg.sender);
    }
    // onlyKeeper call
    if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
        revert Errors.Unauthorized(msg.sender);
    }
    // Check if validator has already exited - do this before checking stake
    if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
        revert Errors.ValidatorForceExited();
    }
    // forced exit always withdraw entire stake of validator
    uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

    // do nothing if InfraredBERA deposit would revert
    uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
        + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
    if (amount < min) return;
    // revert if insufficient balance
    if (amount > address(this).balance) revert Errors.InvalidAmount();

    // todo: verfiy forced withdrawal against beacon roots

    // register new validator delta
    IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

    // re-stake amount back to ibera depositor
    IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
        value: amount
    }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

    emit Sweep(InfraredBERA, amount);
}
```

We can see that the code sets the `uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE`.

Then it runs `if (amount < min) return;`.

However, it isn't depositing into InfraredBERA.sol which is where that condition should be enforced. **In this case it is calling depositor's queue which does not have that requirement**

For reference this is depositor's queue:

```
function queue(uint256 amount) external payable returns (uint256 nonce) {
    // @dev can be called by withdrawor when rebalancing and sweeping
    if (
        msg.sender != InfraredBERA
            && msg.sender != IInfraredBERA(InfraredBERA).withdrawor()
    ) {
        revert Errors.Unauthorized(msg.sender);
    }

    if (amount == 0 || msg.value < amount) revert Errors.InvalidAmount();
    uint256 fee = msg.value - amount;
    if (fee < InfraredBERAConstants.MINIMUM_DEPOSIT_FEE) {
        revert Errors.InvalidFee();
    }
    fees += fee;

    nonce = nonceSlip++;
    slips[nonce] =
        Slip({timestamp: uint96(block.timestamp), fee: fee, amount: amount});
    emit Queue(nonce, amount);
}
```

We can see that as long as `msg.value > InfraredBERAConstants.MINIMUM_DEPOSIT_FEE` then it won't revert and it would run properly.

Hence the `min` that is compared to `amount` in `sweep(pubkey)` should just be `MINIMUM_DEPOSIT_FEE + 1` which is just `1 ether` **and not** `11 ether`(coz `MINIMUM_DEPOSIT=10 ether`).

Impact: This could result in stuck funds due to `sweep` not working properly.

`InfraredBERAWithdraworLite.sol` is a temporary contract that temporarily replaces `InfraredBERAWithdrawor.sol` as BERA chain is yet to enable voluntary withdrawals.

Hence, once berachain enables it, the linked withdrawor contract will be changed to `InfraredBERAWithdrawor.sol` and **all future funds will rightfully be directed there**, then the `keeper` should call `sweep` in the old `Lite` version so that the funds in address(Lite).balance can be retrieved and sent to the depositor contract.

Since `InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE = 11 ether` BERA.

So if the balance is 11 ether - 1 wei then `sweep` will wrongly revert and prevent the funds from being sent to depositor's queue due to the wrong check.

Recommendation: Change `min` to the **depositor queue condition and not the condition that is required for InfraredBERA's** `_deposit` which is a different function that **isn't called by sweep**

```
-   uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
+   uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT_FEE + 1;
```

3.2.27 Setting `feeDivisorShareholders` will result in inaccurate reward calculations

Severity: Medium Risk

Context: InfraredBERA.sol#L247-L250

Summary: The governor can update the `feeDivisorShareholders` value which is used in rewards calculations. This function does not call `compound()`, meaning that rewards are not accurately accrued up to this point. Instead, the retroactive rewards will be accrued based on the newly-updated `feeDivisorShareholders` value.

Finding Description:

`feeDivisorShareholders` is used as the denominator in the fee calculation:

```
    /// @inheritdoc IInfraredBERAFeeReceivor
    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;
        }
    }
```

Since `feeDivisorShareholders` can be updated, either larger or smaller, and the retroactive fees are not accrued prior to updating this value, fees will be accrued based on the updated value.

Impact Explanation:

- Accrued fees will be rewarded incorrectly.

Recommendation: Call `compound()` in the `setFeeDivisorShareholders()` function:

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

3.2.28 Attacker can intentionally skip minting RED token in try/catch block, causing loss of rewards to vault

Severity: Medium Risk

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

Finding Description: Let's look at how BGT rewards are claimed for vaults:

Infrared.sol#L467-L478.

```
    function harvestVault(address _asset) external {
        IInfraredVault vault = vaultRegistry(_asset);
@>      uint256 bgtAmt = _rewardsStorage().harvestVault(
            vault,
            address(_bgt),
            address(ibgt),
            address(voter),
            address(red),
            rewardsDuration()
        );
        emit VaultHarvested(msg.sender, _asset, address(vault), bgtAmt);
    }
```

The function is permissionless, and will execute the implementation `harvestVault` of the rewards library:

RewardsLib.sol#L127-L206.

```
    function harvestVault(
        RewardsStorage storage $,
        IInfraredVault vault,
        address bgt,
        address ibgt,
        address voter,
        address red,
        uint256 rewardsDuration
    ) external returns (uint256 bgtAmt) {
        // Ensure the vault is valid
        if (vault == IInfraredVault(address(0))) {
            revert Errors.VaultNotSupported();
        }

        // Record the BGT balance before claiming rewards
        uint256 balanceBefore = _getBGTBalance(bgt);

        // Get the rewards from the vault's reward vault
        IBerachainRewardsVault rewardsVault = vault.rewardsVault();
        rewardsVault.getReward(address(vault), address(this));

        // Calculate the amount of BGT rewards received
        bgtAmt = _getBGTBalance(bgt) - balanceBefore;

        // If no BGT rewards were received, exit early
        if (bgtAmt == 0) return bgtAmt;

        // Mint InfraredBGT tokens equivalent to the BGT rewards
        IInfraredBGT(ibgt).mint(address(this), bgtAmt);

        // Calculate and distribute fees on the BGT rewards
        (uint256 _amt, uint256 _amtVoter, uint256 _amtProtocol) =
        _chargedFeesOnRewards(
            bgtAmt,
            $.fees[uint256(ConfigTypes.FeeType.HarvestVaultFeeRate)],
            $.fees[uint256(ConfigTypes.FeeType.HarvestVaultProtocolRate)]
        );
        _distributeFeesOnRewards(
            $.protocolFeeAmounts, voter, ibgt, _amtVoter, _amtProtocol
        );

        // Send the remaining InfraredBGT rewards to the vault
        if (_amt > 0) {
            ERC20(ibgt).safeApprove(address(vault), _amt);
            vault.notifyRewardAmount(ibgt, _amt);
        }

        uint256 mintRate = $.redMintRate;
        // If RED token is set and mint rate is greater than zero, handle RED rewards
        if (red != address(0) && mintRate > 0) {
            // Calculate the amount of RED tokens to mint
            uint256 redAmt = bgtAmt * mintRate / RATE_UNIT;
@>          try IRED(red).mint(address(this), redAmt) {
                {
                    // Check if RED is already a reward token in the vault
                    (, uint256 redRewardsDuration,,,,,) = vault.rewardData(red);
                    if (redRewardsDuration == 0) {
                        // Add RED as a reward token if not already added
                        vault.addReward(red, rewardsDuration);
                    }
                }

                // Calculate and distribute fees on the RED rewards
                (_amt, _amtVoter, _amtProtocol) =
                    _chargedFeesOnRewards(redAmt, 0, 0);
                _distributeFeesOnRewards(
                    $.protocolFeeAmounts, voter, red, _amtVoter, _amtProtocol
                );

                // Send the remaining RED rewards to the vault
                if (_amt > 0) {
                    ERC20(red).safeApprove(address(vault), _amt);
                    vault.notifyRewardAmount(red, _amt);
                }
@>          } catch {
                emit RedNotMinted(redAmt);
            }
        }
    }
```

Looking at the try/catch block, if the attempt to mint RED (reward) token fails, an event is emitted and the function successfully executes. An attacker can intentionally send enough gas such that the try block `mint` call fails due to OOG (out of gas) error, which goes into the catch block where only an event is emitted, since only 63/64 gas is forwarded, the `harvestVault` function succeeds.

Impact Explanation: Medium - only RED reward tokens are lost, BGT rewards are still successfully sent.

Likelihood Explanation: Medium - looks like some other factors must still be true for RED token to mint: non-zero RED address, mintRate > 0, mint must succeed. Therefore for the attacker to exploit this, those factors must also be true (otherwise RED token won't be minted anyway), giving a medium likelihood.

Recommendation: I would recommend checking if there is enough gas to execute the mint call beforehand, and reverting if there is not.

3.2.29 It would be impossible to cancel boost for removed validators

Severity: Medium Risk

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

Summary: It would be impossible to cancel boost for removed validators due to the improper validation in `cancelBoosts()`.

Finding Description: After the mitigation of this Spearbit finding - `5.6.14 Missing sanity check on_pubkeysincancelBoosts()andcancelDropBoosts()is inconsistent`, `cancelBoosts()` checks if `pubkey` is an active validator or not.

```
    function cancelBoosts(
        ValidatorStorage storage $,
        address bgt,
        bytes[] memory _pubkeys,
        uint128[] memory _amts
    ) external {
        if (_pubkeys.length != _amts.length) {
            revert Errors.InvalidArrayLength();
        }
        for (uint256 i = 0; i < _pubkeys.length; i++) {
            bytes memory pubkey = _pubkeys[i];
            bytes32 id = keccak256(pubkey);
            if (!$.validatorIds.contains(id)) {
                revert Errors.InvalidValidator(); //@audit can't cancel for removed validator
            }
            if (_amts[i] == 0) revert Errors.ZeroAmount();
            IBerachainBGT(bgt).cancelBoost(pubkey, _amts[i]);
        }
    }
```

But a DoS scenario would exist.

- `queueBoosts()` is called for an active validator.
- After that, that validator has been removed by a governor.
- While calling `cancelBoosts()` to cancel the queued boosts for the validator, it will revert. Also, it's impossible to activate the boosts because activateBoosts() contains the same validation.
- As a result, the queued boosts would be locked until a governor enables the validator again.

Furthermore, this comment says correctly but it seems to be changed during a mitigation.

```
    /**
     * @notice Removes `_amts` from previously queued boosts to `_validators`.
     * @dev `_pubkeys` need not be in the current validator set in case just removed but need to cancel.
     * @param _pubkeys     bytes[] memory The pubkeys of the validators to remove boosts for.
     * @param _amts        uint128[] memory The amounts of BGT to remove from the queued boosts.
     */
    function cancelBoosts(bytes[] memory _pubkeys, uint128[] memory _amts)
        external;
```

A similar scenario exists with `queueDropBoosts()` and `cancelDropBoosts()` as well.

Impact Explanation: The protocol's boost limits woudn't be used properly for active validators.

Likelihood Explanation: Medium because validators could be removed anytime and `queueBoosts()` is managed by a keeper role.

Proof of Concept: None.

Recommendation: Recommend cancelling queued boosts(drop boosts also) before removing a validator.

3.2.30 Register() in InfraBera.sol allows to permanently disable a Validator if the keeper period has passed

Severity: Medium Risk

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

Summary:

`register()` in `InfraBera.sol` allows permanently disabling a Validator, which can be exploited through a withdrawal request if the keeper period has passed.

Finding Description: In `register()` its possible to permanently disable validator, if we make it reach `_exited()` state. After this stage, no more liquid staking deposits can be added to that same validator.

`exited` state is reached when all the liquid stake for X validator is withdrawn.

```
    function register(bytes calldata pubkey, int256 delta) external {
        if (msg.sender != depositor && msg.sender != withdrawor) {
            revert Errors.Unauthorized(msg.sender);
        }
        if (_exited[keccak256(pubkey)]) {
@>>         revert Errors.ValidatorForceExited();
        }

        uint256 stake = _stakes[keccak256(pubkey)];
        if (delta > 0) stake += uint256(delta);
        else stake -= uint256(-delta);
        _stakes[keccak256(pubkey)] = stake;

        if (delta > 0 && !_staked[keccak256(pubkey)]) {
            _staked[keccak256(pubkey)] = true;
        }
        // only 0 if validator was force exited
        if (stake == 0) {
            _staked[keccak256(pubkey)] = false;
@>>         _exited[keccak256(pubkey)] = true;
        }

        emit Register(pubkey, delta, stake);
    }
```

This is possible to be exploited, in case the keeper is not active for some time and the _enoughtime() check allows for an arbitrary party to execute the withdrawal.

```
            if (!kpr && !_enoughtime(r.timestamp, uint96(block.timestamp))) {
                revert Errors.Unauthorized(msg.sender);
            }
```

An adversary can choose a validator with a small liquid staking amount (this can be a freshly added validator) and exit all its capital when enough withdrawal amounts are queued.

Impact Explanation: Permanently disabling validator, denying it from receiving further liquid staking capital deposits.

Recommendation: In `InfraredBera.sol` add a way for a trusted party to enable again a validator, that has exited.

3.2.31 Rewards claiming is bricked due to the onReward() hook when Bera governance hasn't approved RewardVault for BGT transfers yet

Severity: Medium Risk

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

Summary: Rewards claiming can be temporary bricked/DoSed if Bera governance hasn't approved a newly created RewardVault to transfer BGT tokens.

Finding Description: The whole process is explained here `https://docs.berachain.com/learn/governance/rewardvault#reward-vault-governance`:

> While creating a Reward Vault is permissionless, for it to receive $BGT emissions from validators, it must be whitelisted through a governance proposal. This process ensures community oversight and alignment over projects joining the Proof-of-Liquidity (PoL) ecosystem.

BGT tokens by default are not transferable and only whitelisted senders can send them. Reference: BGT.sol#L328-L339.

```
    function transfer(
        address to,
        uint256 amount
    )
        public
        override(IERC20, ERC20Upgradeable)
@>>     onlyApprovedSender(msg.sender)
        checkUnboostedBalance(msg.sender, amount)
        returns (bool)
    {
        return super.transfer(to, amount);
    }
```

However RewardVaults need to send those tokens when calling getReward(), because their rewardToken is BGT: RewardVault.sol#L103.

```
    function initialize(
        address _beaconDepositContract,
@>>     address _bgt,
        address _distributor,
        address _stakingToken
    )
        external
        initializer
    {
        __FactoryOwnable_init(msg.sender);
        __Pausable_init();
        __ReentrancyGuard_init();
@>>    __StakingRewards_init(_stakingToken, _bgt, 7 days);
```

Thus they need to be whitelisted by `Bera governance`, which could take some time. This however **denies claiming any other rewards** from `InfraredVault`, which is linked to a `Bera RewardVault` that hasn't been approved yet.

In `InfraredVault`, we use `onReward()` hook to claim rewards from `RewardsVault` from Bera, however we enforce it on every `getReward()` call..

```
    function getReward() public {
        getRewardForUser(msg.sender);
    }

    function getRewardForUser(address _user)
        public
        nonReentrant
        updateReward(_user)
    {
@>>     onReward();
        uint256 len = rewardTokens.length;
        for (uint256 i; i < len; i++) {
            address _rewardsToken = rewardTokens[i];
            uint256 reward = rewards[_user][_rewardsToken];
            if (reward > 0) {
                rewards[_user][_rewardsToken] = 0;
                (bool success, bytes memory data) = _rewardsToken.call(
                    abi.encodeWithSelector(
                        ERC20.transfer.selector, _user, reward
                    )
                );
                if (success && (data.length == 0 || abi.decode(data, (bool)))) {
                    emit RewardPaid(_user, _rewardsToken, reward);
                } else {
                    continue;
                }
            }
        }
    }
```

```
    function onReward() internal override {
        IInfrared(infrared).harvestVault(address(stakingToken));
    }
```

`harvestVault()` is supposed to harvest BGT rewards generated in the bera `RewardsVault` and distribute `IBGT` and `RED` tokens to `InfraredVault`.

```
 function harvestVault() external returns (uint256 bgtAmt) {
        if (vault == IInfraredVault(address(0))) {
            revert Errors.VaultNotSupported();
        }
        uint256 balanceBefore = _getBGTBalance(bgt);

        IBerachainRewardsVault rewardsVault = vault.rewardsVault();
@>>     rewardsVault.getReward(address(vault), address(this));
```

But `rewardsVault.getReward()` requires the rewardsVault to be approved to send BGT, which in a recently created vault it might not be the case. This would result in a DoS.

Impact Explanation: Its also possible that rewards which do not come from `Bera RewardsVault` have been donated/distributed to the `InfraredVault` (for example via Infrared.addIncentives()), thus this will temporary deny their claiming until `Bera RewardsVault` has been whitelisted to transfer BGT.

Likelihood Explanation: This can happen for recently-created `InfraredVaults`.

Recommendation: In `harvestVault()` check if the RewardsVault is whitelisted for BGT transfer. If not then do not harvest.

~~~
+ if(!bgt.isWhitelistedSender(rewardsVault) return;





### Malicious actors can dilute the staking rewards to a longer timeframe
<!--
Number: 445
Candinacode repository status: confirmed
Hyperlink: [Issue 445](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/findings/445)
Labels: [Primary]
Fixed on: [None]
-->


**Severity:** Medium Risk

**Context:** [MultiRewards.sol#L313-L326](https://cantina.xyz/code/ac5f64e6-3bf2-4269-bbb0-4bcd70425a1d/src/co⌋
↪  re/MultiRewards.sol#L313-L326)

**Summary:** Infrared allows any user to add incentives in the form of whitelisted tokens to vaults on
↪  Infrared. However, adding incentives while other incentives are already live dilutes the total reward over a
↪  longer timeframe, which can be exploited by malicious actors.

**Finding Description:** Consider the following test case added to `MultiRewardsForkTest.t.sol`:

```Solidity
    function testDisruptedRewards() public {
        uint cycleLength = 3600;

        vm.startPrank(infraredGovernance);
        // whitelist usdc
        infrared.updateWhiteListedRewardTokens(address(usdc), true);
        infrared.addReward(address(stakingToken), address(usdc), cycleLength);
        // whitelist usdt
        infrared.updateWhiteListedRewardTokens(address(usdt), true);
        infrared.addReward(address(stakingToken), address(usdt), cycleLength);
        vm.stopPrank();

        vm.startPrank(alice);
        // add usdc
        infrared.addIncentives(address(stakingToken), address(usdc), 1e20);
        // add usdt
        infrared.addIncentives(address(stakingToken), address(usdt), 1e20);
        vm.stopPrank();

        // stake
        stakeAndApprove(alice, 1e18);

        // add zero incentives every second
        uint cycles = cycleLength;
~~~

```
        for (uint i = 0; i < cycles; ++i) {
            skip(1);

            // add 1 token of incentives to usdt
            deal(address(usdt), alice, 1);
            vm.prank(alice);
            infrared.addIncentives(address(stakingToken), address(usdt), 1);
        }

        // At this point the earnings should be the same for USDC and USDT
        uint256 earningsUsdc = lpVault.earned(alice, address(usdc));
        uint256 earningsUsdt = lpVault.earned(alice, address(usdt));
        console2.log("earningsUsdc", earningsUsdc);
        console2.log("earningsUsdt", earningsUsdt);
        assertGt(earningsUsdc, earningsUsdt);
    }
```

In the showcased test, we can observe two rewards - `USDC` and `USDT`. `1e20` of both tokens were added for the same `stakingToken`, and both have the same duration of `3600`. Since Alice is the only one staking in the vault, they should receive all tokens after `3600` seconds have passed (minus a small allocation due to the 1 wei stake of the vault).

However, if a malicious user were to add incentives of 1 wei every second after the start, the reward would be diluted due to the line:

```
rewardData[_rewardsToken].rewardRate = totalAmount / rewardData[_rewardsToken].rewardsDuration;
```

This causes the reward rate to be updated to the total amount (which is current amount left) divided by the reward duration.

Upon running the provided test, we can see the following output:

```
earningsUsdc 99999999999999993600
earningsUsdt 63217165910990835140
```

This means that after 3600 seconds, the whole reward of `USDC` was earned, while only around 63% of the `USDT` reward was earned (although practically no reward was added), which is clearly undesired behavior.

Impact Explanation: Medium. Time to gain the intended reward can be arbitrarily increased by malicious users.

Likelihood Explanation: Medium. While the attacker might not profit from this behavior, it is fairly cheap to perform and can also occur naturally.

Recommendation: Consider adding new rewards to a buffer of rewards, which would not influence the current reward period. Once the current reward period is finished, a new one can start with new rewards allocated from the reward buffer.

3.2.32 InfraredVault does not claim frequently-enough the rewards from BeraRewardVault

Severity: Medium Risk

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

Summary:

`InfraredVault` **only** claims rewards from `BeraRewardVault` when someone tries to get their rewards from `InfraredVault`. This is not ideal as it does not keep big accuracy of the reward distribution.

Finding Description: The problem is that there is `rewardDuration` period both on `BeraRewardVault` and `InfraredVault`. So we have:

- `InfraredVault` waiting `rewardDuration` period on `BeraRewardVault`.
- `InfraredVault` stakers waiting for `rewardDuration` on `InfraredVault`.

  Disclaimer: they are not really "waiting", but rewardDuration is crucial, as when rewards are send to a vault a new rate is calculated based on the rewards and the rewardDuration is set.

This means that the less frequently rewards are transferred from `BeraRewardVault` -> `InfraredVault`, the less amounts will be "distributed" to stakers in `InfraredVault` for the same period of time.

Ideally we could call this hook onReward() to harvest rewards, on every intraction with the `InfraredVault`.

```
    /**
     * @notice hook called after the reward is claimed to harvest the rewards from the berachain rewards vault
     */
    function onReward() internal override {
        IInfrared(infrared).harvestVault(address(stakingToken));
    }
```

Impact Explanation: If rewards are not frequently send from `BeraRewardVault` -> `InfraredVault`, some Stakers could receive less amounts than they should, given that they withdraw, because for the period they were staking, they would've received more, if rewards from `BeraRewardVault` were distributed more frequently.

Recommendation: Its recommended to enforce `onReward()` hook when `staking` and when `withdrawing` as well.

3.2.33 BGT tokens will be forever locked in Infrared.sol as there is no way to redeem/withdraw them

Severity: Medium Risk

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

Summary: There is no mechanism to withdraw or redeem BGT tokens from `Infrared.sol` contract, thus they are stuck forever there.

Finding Description: BGT tokens are received from RewardVaults and are minted 1:1 with IBGT tokens. However there isn't any way to get out/redeem BGT tokens for BERA from `Infrared.sol`.

We only harvest excessive amounts of BGT for BERA:

```
    function harvestBase(address bgt, address ibgt, address ibera)
        external
        returns (uint256 bgtAmt)
    {
        uint256 minted = IInfraredBGT(ibgt).totalSupply();
        uint256 bgtBalance = _getBGTBalance(bgt);
        // @dev should never happen but check in case
        if (bgtBalance < minted) revert Errors.UnderFlow();

@>>     bgtAmt = bgtBalance - minted; // only excessive amount is gonna be redeemed.
        if (bgtAmt == 0) return 0;

        // Redeem BGT for BERA and send to InfraredBERA receivor
        // No fee deduction needed here as fees will be handled by
        // subsequent harvest calls through the InfraredBERA receiver's logic

        IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);
    }
```

This ensures 1:1 -> BGT reserves : IBGT supply , however there isn't any way for the IBGT holders to get BERA/redeem for BGT. For reference if the staker, which received IBGT tokens, decided to directly stake into RewardsVault on Bera instead of the linked InfraredVault, they would've gotten the BGT tokens, and would've been able to redeem them for bera..

If the goal is to maintain PEG: Its likely that **PEG would be hard to be kept 1:1 this way**, as when holders of IBGT cannot redeem 1:1 for BGT/Bera, they are relying on market speculation to maintain the price, but not on algorithmic stabilization. And if there is a selling pressure, there isn't an arbitrage opportunity that will help maintain the PEG.

Otherwise even if PEG is not crucial, we still have the issue of never being able to get out the BGT of the `Infrared.sol`, as there isn't such logic.

Impact Explanation:

- Unable to withdraw/redeem IBGT for BGT/BERA. Or get out BGT from the Infrared.sol contract.
- Potential pegging issues.

Recommendation: Add a way to redeem/withdraw IBGT/BGT for BERA.

3.2.34 InfraredBera's initial deposit limit is too low causing the staked Bera to be returned, and the validator to be marked as exited

Severity: Medium Risk

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

Summary: Currently InfraredBera's depositor checks whether the initial deposit is higher than 10K Bera, if it is, it allow it to be staked within Bera's deposit contract intended for validator staking..

The problem is that the minimum floor for a validator is 250K Bera, not 10K. This will cause any staked amount below 250k to be returned to the withdrawor contract.

This will cause the validator to be marked as "force-exited" and will prompt the creation of new validator keys, as that validator will no longer be able to function within Infrared's system do to it being marked as `exited`.

Finding Description: As per Berachain's PoL documentation:

- `https://docs.berachain.com/learn/pol/bgtmath#validator-selection`.
- `https://docs.berachain.com/nodes/guides/validator#run-a-validator-node-on-berachain`.
- `https://docs.berachain.com/learn/pol/faqs#validator-requirements-operations`.

"Stake limitations per validator: Minimum: 250,000 BERA. Maximum: 10,000,000 BERA".

A validator needs 250K Bera minimum. If a validator's stake is too low, it will be returned to the address supplied in the credentials -> in this case, the Withdrawor address.

As it can be seen from the Bera Depositor contract, during the initial deposit of a validator, the minimum limit imposed is 10k Bera:

```
uint256 public constant INITIAL_DEPOSIT = 10000 ether;

    // Add first deposit validation
        if (currentOperator == address(0)) {
            if (amount != InfraredBERAConstants.INITIAL_DEPOSIT) {
                revert Errors.InvalidAmount();
            }
        } else {
            // Verify subsequent deposit requirements
            if (currentOperator != operator) {
                revert Errors.UnauthorizedOperator();
            }
        }
```

Considering the fact that besides the Keeper, users are able to execute deposits to Berachain once the forced minimum delay passes, it means that initial stakes of validators in the range of 10K -249.9K BERA are plausible to be executed not only by trusted system entities (keeper), but by arbitrary users as well.

```
// @dev allow user to force stake into infrared validator if enough time has passed
            // TODO: check signature not needed (ignored) on second deposit to pubkey (think so)
            if (!kpr && !_enoughtime(s.timestamp, uint96(block.timestamp))) {
                revert Errors.Unauthorized(msg.sender);
            }
```

If the initial stake sent to the Berachain Depositor contract is less than the minimum (250k), it will be returned to the Withdrawor Lite contract. In here we have a `sweep()` function which serves to re-deposit, i.e. re-queue the returned amount in the depositor contract, assuming that it was due to a force-exit of the validator..

```
function sweep(bytes calldata pubkey) external {
      // only callable when withdrawals are not enabled
      if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
          revert Errors.Unauthorized(msg.sender);
      }
      // onlyKeeper call
      if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
          revert Errors.Unauthorized(msg.sender);
      }
      // Check if validator has already exited - do this before checking stake
      if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
          revert Errors.ValidatorForceExited();
      }
      // forced exit always withdraw entire stake of validator
      uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

      // do nothing if InfraredBERA deposit would revert
      uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
          + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
      if (amount < min) return;
      // revert if insufficient balance
      if (amount > address(this).balance) revert Errors.InvalidAmount();

      // todo: verfiy forced withdrawal against beacon roots

      // register new validator delta
      IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

      // re-stake amount back to ibera depositor
      IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
          value: amount
      }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

      emit Sweep(InfraredBERA, amount);
  }
```

Due to the following conditions, there are several problems which arise:

- `InfraredBERAConstants.INITIAL_DEPOSIT` is a constant, marked at 10K BERA which can't be adjusted by Governance, so that it matches the current deposit limit, OR to be able to be adjusted in the future if a validator's initial stake needs to go above 250K BERA in order to enter the active validator set..
- The current mechanics, would cause the return of most initial deposits made as they don't suffice Berachain's minimum amount criteria, which means that each validator's stake which was returned would have to be re-deposited.
- Each validator's stake which was returned, will no longer be able to be used, prompting the inclusion of a new validator and the generation of a new key/signature..

Regarding the last point, once an amount of Bera was returned to the Withdrawor contract, we can see that it's registered within Infrared:

```
      // register new validator delta
      IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));
```

And since said amount includes the whole virtual balance of the validator, their status will be marked as `exited`, and that validator will no longer be able to be utilized as the function checks if the validator is marked as exited so no new deposits are able to be made to that pubkey:

```
function register(bytes calldata pubkey, int256 delta) external {
        if (msg.sender != depositor && msg.sender != withdrawor) {
            revert Errors.Unauthorized(msg.sender);
        }
        if (_exited[keccak256(pubkey)]) {
            revert Errors.ValidatorForceExited();
        }
        // update validator pubkey stake for delta
        uint256 stake = _stakes[keccak256(pubkey)];
        if (delta > 0) stake += uint256(delta);
        else stake -= uint256(-delta);
        _stakes[keccak256(pubkey)] = stake;
        // update whether have staked to validator before
        if (delta > 0 && !_staked[keccak256(pubkey)]) {
            _staked[keccak256(pubkey)] = true;
        }
        // only 0 if validator was force exited
        if (stake == 0) {
            _staked[keccak256(pubkey)] = false;
            _exited[keccak256(pubkey)] = true;
        }

        emit Register(pubkey, delta, stake);
    }
```

Impact Explanation: Most deposits to Bera will fail due to not satisfying the minimum stake threshold, prompting the creation of a new validator and subsequently pubkey/signature due to it being marked as `exited`.

Likelihood Explanation: Considering that the threshold is only 10K and arbitrary users can make deposits to Bera's deposit contract, this is bound to happen frequently.

Proof of Concept:

- A 10K BERA Initial deposit is made to Bera's deposit contract by an arbitrary user as an initial validator stake;
- Due to the deposit not satisfying the minimum amount criteria, the validator isn't included in the validator set, and the BERA-in-question is returned to the address outlined in the credentials (Withdrawor contract);.
- Keeper calls `sweep()` on the WithdraworLite contract in order to re-queue the BERA for a subsequent deposit.
- Following the above action, the validator is marked as `exited` and can no longer be used, prompting the need of a new validator, and the creation of new validator keys.
- This is bound to happen any and every time with a new validator due to the deposit threshold being very low.

Recommendation: Make the deposit threshold modifiable by Governance, to adjust it to Bera's criteria and for any future increases that might need to be applied due to Bera raising the threshold OR in order to have more stake than the last validators in the active validator set (As 250k Bera is not enough, you also need to have more stake than the last validator in the active set)..

3.2.35 Funds can be permanently locked when keeper is inactive while validator is forced exit

Severity: Medium Risk

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

Finding description: According to current implementation of BeaconKit, when voluntary withdrawal is not yet enabled, validators can be forced exit when they are kicked out of the validator set (meaning their stakes are the least and the number of eligible validators exceeds the cap)..

See: `https://github.com/berachain/beacon-kit/blob/main/state-transition/core/README.md`.

When validators are forced exit, their `exitEpoch` is set and they can never become active again. That is, in order to participate in consensus again, they need to issue a new public key with a new deposit. (See: SetExitEpoch and isActive).

In addition, when forced exit, their deposit balance are automatically sent to designated withdrawer address as specified in withdrwal credentials set in first deposit..

All Infrared's validators are onboarded with the same withdrawer address being set to **InfraredBERAWithdrawor** Therefore, when validators are forced exit, deposit balances are sent to withdrawor contract. (Note that Lite version is used when voluntary withdrawal is not yet enabled)..

Infrared then handles the forced exit by invoking `sweep` function (only callable by keeper)..

```
// File: InfraredBERAWithdraworLite.sol
    function sweep(bytes calldata pubkey) external {
        // only callable when withdrawals are not enabled
        if (IInfraredBERA(InfraredBERA).withdrawalsEnabled()) {
            revert Errors.Unauthorized(msg.sender);
        }
        // onlyKeeper call
        if (!IInfraredBERA(InfraredBERA).keeper(msg.sender)) {
            revert Errors.Unauthorized(msg.sender);
        }
        // Check if validator has already exited - do this before checking stake
        if (IInfraredBERA(InfraredBERA).hasExited(pubkey)) {
            revert Errors.ValidatorForceExited();
        }
        // forced exit always withdraw entire stake of validator
        uint256 amount = IInfraredBERA(InfraredBERA).stakes(pubkey);

        // do nothing if InfraredBERA deposit would revert
        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (amount < min) return;
        // revert if insufficient balance
        if (amount > address(this).balance) revert Errors.InvalidAmount();

        // todo: verfiy forced withdrawal against beacon roots

        // register new validator delta
        IInfraredBERA(InfraredBERA).register(pubkey, -int256(amount));

        // re-stake amount back to ibera depositor
        IInfraredBERADepositor(IInfraredBERA(InfraredBERA).depositor()).queue{
            value: amount
        }(amount - InfraredBERAConstants.MINIMUM_DEPOSIT_FEE);

        emit Sweep(InfraredBERA, amount);
    }
```

The `sweep` function serves two purposes:

1. Mark the validator with specified pubkey as exited, so it no longer available for deposit in Depositor contract..
2. Redeposit withdrawn amount to Depositor..

Within deposit flow (InfraredBERADEpositor.execute), Infrared tries to account for the case wherein keeper is inactive (for 7 days) by allowing manual execution to proceed remaining deposit slips..

There are many checks and validations in place to ensure that a new deposit only goes to legitimate validators, including that target validator is not being forced exit..

However, if we consider a situation wherein keeper is inactive. It is possible that some validators might be forced exit during this period.. This creates a problem as validator's status is not updated to match that of beacon chain state via `sweep` function, so, the deposit can still go in through `InfraredBERADEpositor.execute` to this exited validator through manual execution..

```
// File: InfraredBERADepositor.sol
function execute(bytes calldata pubkey, uint256 amount) external {
    // ...
    // ... snipped
    // ...
    IBeaconDeposit(DEPOSIT_CONTRACT).deposit{value: amount}(
        pubkey, credentials, signature, operator
    );

    // register update to stake
    IInfraredBERA(InfraredBERA).register(pubkey, int256(amount)); // safe as max fits in uint96
    // @audit: this check within register function would pass because validator's status is not updated to exit
    ↪  yet via sweep function

    // sweep fee back to keeper to cover gas
    if (fee > 0) SafeTransferLib.safeTransferETH(msg.sender, fee);

    emit Execute(pubkey, _nonce, nonce, amount);
}
```

The beacon chain will pickup the deposit and increase validator's balance but it will never get included in validator set because it's already exited..

And because there is only one way to withdraw the balance is via getting kicked out of validator set, the deposited funds might get stuck forever if Berachain were to follow ETH Beacon Chain, validators might not be able to withdraw this balance as exited validators are not eligible for withdrawal request..

See: Lodestar, Lighthouse.

There is also another scenario where this could happen, if the keeper doesn't sync validator's status with beacon chain, keeper could fire a deposit transaction simultaneously when the target validator is forced exit..

Example Scenario:

1. The keeper is inactive, manual deposit is eligible.
2. A validator is forced to exit due to being kicked out of the validator set during this inactive period..
3. Due to inactive keeper, sweep is not called to update the validator's status..
4. A user manually executes a deposit to the just-exited validator via InfraredBERADepositor.execute..
5. The beacon chain accepts the deposit and increases the validator's balance, but the validator remains exited and cannot proceed withdrawals..
6. The deposited funds are permanently locked in the validator's balance while also not being included in validator set..

Impact: Deposits made to validators that have been forced to exit but not marked as exited in the InfraredBERA contract will be locked indefinitely.

Rationale for severity:

- Likelihood: low due to required preconditions: keeper is in active, validator is forced exit during inactive period.
- Impact: high due to potential permentant funds stuck.

Hence, Medium severity..

Recommened Mitigations: The best mitigation would be to utilize beacon state root to determine validator's status before deposit.. Simply strictly only allow keeper to execute deposit could mitigate the issue but at a complete trust on keeper not being malicious..

3.2.36 Potential Reward Calculation Issue in updateReward Function

Severity: Medium Risk

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

Summary: The updateReward function has a potential issue where users may receive fewer rewards than expected if multiple periodFinish intervals pass without any staking activity or reward distribution. This is due to the function only updating rewards for a single periodFinish interval, leading to inaccurate reward calculations.

Finding Description: The updateReward function updates the reward data for a given account by iterating through the rewardTokens array. It updates the rewardPerTokenStored and lastUpdateTime for each token. If an account is provided, it also updates the rewards and userRewardPerTokenPaid mappings for that account. However, if multiple periodFinish intervals pass without any staking activity or reward distribution, the function will only update the rewards for the most recent periodFinish, causing users to receive fewer rewards for the time that has passed.

Impact Explanation: The impact of this issue is that users may receive fewer rewards than they are entitled to. This can lead to a loss of trust in the system and potential financial losses for users who rely on the rewards as part of their investment strategy. The issue is particularly problematic in scenarios where there is low staking activity or infrequent reward distributions, as the discrepancy between the actual and calculated rewards can grow over time.

Likelihood Explanation: The likelihood of this issue occurring depends on the frequency of staking activity and reward distributions. In systems with high staking activity and frequent reward distributions, the issue is less likely to occur. However, in systems with low staking activity or infrequent reward distributions, the likelihood increases. Given that staking activity can vary significantly over time, this issue has a moderate likelihood of occurring in real-world scenarios.

Proof of Concept (if required): Consider a scenario where 3 periodFinish intervals pass without any staking activity or reward distribution:

First periodFinish Interval:

The updateReward function updates the reward data for the first interval.

Users receive rewards for the first interval.

Second periodFinish Interval:

No staking activity or reward distribution occurs.

Third periodFinish Interval:

The updateReward function updates the reward data for the third interval.

Users receive rewards for the third interval.

Users receive rewards only for the first and third interval, missing out on rewards for the second interval.

This results in users receiving fewer rewards than they should have for the time that has passed.

Recommendation (optional:

).

3.2.37 Compounding logic doesn't account for pending rewards that need to be sent to the Receivor resulting in a share disbalance

Severity: Medium Risk

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

Summary: Although this problem persists in InfraredBera withdrawals as well, they will be omitted in this report as they're considered out of scope..

Whenever a user wants to stake their BERA in exchange for iBERA, they can do so via InfraredBERA.sol's `mint()` function. In each `mint()` call the function subsequently calls `compound()` to make sure that the yield earned from Execution Level rewards is accounted for in the deposits, upon which the number of shares minted is also heavily dependent..

The problem is that this function / compounding logic doesn't account for BGT rewards "pending" in the Infrared.sol contract, and doesn't "upkeep" it via calling `harvestBase()`..

Finding Description: InfraredBERA's FeeReceivor receives incentives via two different mechanisms:

- Each time that an auction takes place in the BribeCollector, the FeeReceivor takes parts of the WBERA (withdrawn and transferred as BERA).
- Each time that there is a surplus of BGT in the Infrared.sol contract, a permissionless upkeep action which is performed by calling `harvestBase()`.

The fundamental logic in the share minting in InfraredBERA.sol relies on compounding any outstanding FeeReceivor balance in order to account for any pending deposit in the share calculation. The problem is that the `compound()` mechanism never takes into account any surplus in the Infrared.sol contract which should be transferred to the FeeReceivor and can significantly impact the share ratio and the number of minted shares.

As the above mechanism isn't atomic, and is heavily dependent on voluntary action, i.e. `harvestBase()` being frequently called so that it's properly upkept -> the shares that users will get minted won't always reflect the compound yield earned from EL to be accounted for in the deposits.

When a user calls `mint()`, it first compounds the rewards sent to the FeeReceivor contract in order to include them in the total deposits:

```
 function mint(address receiver)
        public
        payable
        returns (uint256 nonce, uint256 shares)
    {
        // compound yield earned from EL rewards first
        compound();

...

shares = (d != 0 && ts != 0) ? ts * amount / d : amount;
```

Here's the `compound()` logic:

```
function compound() public {
        IInfraredBERAFeeReceivor(receivor).sweep();
    }

function sweep() external returns (uint256 amount, uint256 fees) {
        (amount, fees) = distribution();
        // do nothing if InfraredBERA deposit would revert
        uint256 min = InfraredBERAConstants.MINIMUM_DEPOSIT
            + InfraredBERAConstants.MINIMUM_DEPOSIT_FEE;
        if (amount < min) return (0, 0);

        // add to protocol fees and sweep amount back to ibera to deposit
        if (fees > 0) shareholderFees += fees;
        IInfraredBERA(InfraredBERA).sweep{value: amount}();
        emit Sweep(InfraredBERA, amount, fees);
    }
```

In case an outstanding amount above 11 ether is present in the contract, it will be deposited in the InfraredBera:

```
    function sweep() external payable {
        if (msg.sender != receivor) {
            revert Errors.Unauthorized(msg.sender);
        }
        _deposit(msg.value);
        emit Sweep(msg.value);
    }
```

The current mechanism never accounts for any surplass BGT balance (rewards) outstanding in the Infrared.sol contract which haven't been forwarded to the FeeReceivor..

If such a balance exists, `harvestBase()` needs to be called so that the balance can be forwarded:

```
function harvestBase(address bgt, address ibgt, address ibera)
     external
     returns (uint256 bgtAmt)
 {
     uint256 minted = IInfraredBGT(ibgt).totalSupply();
     uint256 bgtBalance = _getBGTBalance(bgt);
     if (bgtBalance < minted) revert Errors.UnderFlow();

     bgtAmt = bgtBalance - minted;
     if (bgtAmt == 0) return 0;

     // Redeem BGT for BERA and send to InfraredBERA receivor
     IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);
 }
```

Relying on non-atomic logic that `harvestBase` should be voluntarily called could de-incentivize users as the number of shares that they would receive post-harvestBase call could be less than the number they'd receive prior to it..

Bribe collections made as a fee from BribeCollector auctions are excluded, as they aren't rewards which are being accumulated constantly and are based on an auction event..

In the PoC section below, there are two opposing examples of the share difference between two cases outlining pending rewards in the Infrared.sol, as well as no pending rewards due to a `harvestBase()` call prior to it.

In order to completely account for all compounded rewards in the Infrared ecosystem, `harvestBase()` needs to be part of the `compound()` logic.

Impact Explanation: Minted shares won't reflect any pending fees in the Infrared.sol contract, resulting in the minting of more shares than the caller should receive.

Likelihood Explanation: Each time that there is a surplus of BGT in the Infrared.sol contract, there is a possibility that the user receives more shares than they actually should.

Proof of Concept: For the purpose of this example, let's take into situation two opposing scenarios when it comes to the difference in the shares minted.

Scenario 1 (No `harvestBase()` call):

- There's a surplus of 250 BGT pending in the Infrared.sol contract;
- iBERA total supply: 10_000e18;
- iBera deposits: 11_000e18;
- FeeReceivor balance: 0;
- User decides to mint shares equivalent to a 1000e18 BERA deposit:
- shares = 10_000e18 * 1_000e18 / 11_000e18 = 909.09e18;

Scenario 2 (`harvestBase()` is called prior to the mint call):

- There's a surplus of 250 BGT pending in the Infrared.sol contract, but `harvestBase()` is called and said balance is redeemed and transferred to the FeeReceivor contract;
- iBERA total supply: 10_000e18;
- iBera deposits: 11_000e18;
- FeeReceivor balance: 250e18;
- Due to the outstanding FeeReceivor balance (for the simplicity of the example, we'll exclude shareholderFees), 249e18 BERA are added to the deposits;
- New iBera deposits balance: 11_249;
- shares = 10_000e18 * 1_000e18 / 11_249e18 = 888.96e18 shares;

There's a difference of 20.13e18 shares between the two cases;

Recommendation: Make sure that the compound yield earned from EL is properly calculated in each share minting operation by calling Infrared.sol's `harvestBase()` in each `compound()` call..

3.2.38 Potential Reward Manipulation by Front-Running _notifyRewardAmount

Severity: Medium Risk

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

Summary: A user can potentially front-run the _notifyRewardAmount function to stake tokens just before the reward period is updated, capturing the remaining rewards from the current period. This manipulation occurs because the periodFinish and rewardRate are updated in a way that allows users to exploit the timing of their staking actions.

This allows the user to receive the maximum of the current period.

Finding Description: The issue arises due to the way the _notifyRewardAmount function updates the reward distribution parameters (rewardRate and periodFinish). Here's how the manipulation works:

Front-Running Opportunity:

When the owner calls _notifyRewardAmount to update the rewards, the function calculates the new rewardRate and extends the periodFinish based on the remaining time in the current period.

A user can monitor the mempool for the _notifyRewardAmount transaction and front-run it by staking tokens just before the transaction is executed.

Capturing Residual Rewards:

By staking before the _notifyRewardAmount call, the user can capture the remaining rewards from the current period. This is because the rewardRate and periodFinish are recalculated based on the remaining time and the new reward amount, effectively allowing the user to claim rewards from the current period before it is updated.

Impact on Fairness:

This manipulation breaks the fairness of the reward distribution system, as users who front-run the transaction can unfairly claim a larger share of rewards compared to other stakers.

Then new period time comes in.

3.2.39 By burning IBGT(potentially as little as 1 wei), anyone can temporarily DOS harvestBase

Severity: Medium Risk

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

Summary: Berachain's BGT redeem function includes a safeguard (checkUnboostedBalance) to prevent redeeming more tokens than the user's unboosted balance (unboostedBalance = BGTBalance - (boost + queuedBoost)). In harvestBase, the protocol attempts to redeem BGTBalance - IBGTSupply to avoid reverts. However, if the IBGT supply (IBGTSupply) is burned below the current boosts balance (boost + queuedBoost), the redeem operation will fail. Since the boosts balance is always maximized to earn rewards, even burning a minimal amount of IBGT (as little as 1 wei) can trigger this issue. This results in harvestBase reverting until the keeper executes dropBoost, which is delayed by 8191 blocks. Consequently, critical functions like addValidator, removeValidator, and replaceValidator, which depend on harvestBase, become DOSed.

Finding Description: The BGT redeem function in Berachain uses the checkUnboostedBalance modifier to ensure that the amount being redeemed does not exceed the user's unboosted balance: unboostedBalance = user's BGTBalance - (boost + queuedBoost).

To prevent harvestBase from reverting, Infrared redeems the amount BGTBalance - IBGTSupply:

```
    function harvestBase(
        address bgt,
        address ibgt,
        address ibera
    ) external returns (uint256 bgtAmt) {
        uint256 minted = IInfraredBGT(ibgt).totalSupply();
        uint256 bgtBalance = _getBGTBalance(bgt);
        // @dev should never happen but check in case
        if (bgtBalance < minted) revert Errors.UnderFlow();

++++    bgtAmt = bgtBalance - minted;
        if (bgtAmt == 0) return 0;

        IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);
    }
```

This logic assumes that the redeem operation will always pass because the redeem amount (BGTBalance - IBGTSupply) should never exceed the unboosted balance (BGTBalance - (boost + queuedBoost)). However, this assumption breaks if the IBGT supply is burned, resulting in IBGTSupply being less than (boost + queuedBoost).

The boost + queuedBoost (referred to as boostsBalance) is capped at the IBGT supply to prevent excessive boosts:

```
    function queueBoosts(
        ValidatorStorage storage $,
        address bgt,
        address ibgt,
        bytes[] memory _pubkeys,
        uint128[] memory _amts
    ) external {
        ...
        if (
            _totalBoosts >
            IInfraredBGT(ibgt).totalSupply() -
                (IBerachainBGT(bgt).boosts(address(this)) +
                    IBerachainBGT(bgt).queuedBoost(address(this)))
        ) {
            revert Errors.BoostExceedsSupply();
        }
        ...

    }
```

The protocol assumes that IBGTSupply > boostsBalance, ensuring harvestBase works as intended. But if IBGT tokens are burned such that IBGTSupply becomes less than boostsBalance, harvestBase will attempt to redeem more tokens than the unboosted balance, causing a revert.

The critical point is that the amount of IBGT that must be burned to trigger this revert is minimal: burnAmount = IBGTSupply - boostsBalance + 1.

Given that the boosts balance is maximized to earn rewards, even burning a negligible amount (as little as 1 wei) can result in harvestBase failing.

Once harvestBase starts reverting, the keeper must execute a dropBoost operation to reduce the boosts balance. However, this operation has an enforced delay of 8191 blocks, during which harvestBase will continue to fail. Any functions that rely on harvestBase—such as addValidator, removeValidator, and replaceValidator—will also fail. This creates a significant operational disruption, as the protocol cannot perform essential validator management until the boosts balance is dropped or IBGT supply increases.

PoC:

- Create a file called WhitehatTest.t.sol in tests/unit/core/infrared.
- Paste the following:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import {Helper, IInfrared, InfraredVault, RED} from "./Helper.sol";
import "tests/unit/mocks/MockERC20.sol";
import {BGTStaker} from "@berachain/pol/BGTStaker.sol";
import "@berachain/pol/rewards/RewardVaultFactory.sol";
import {IRewardVault as IBerachainRewardsVault} from "@berachain/pol/interfaces/IRewardVault.sol";
import {IInfraredBERAFeeReceivor} from "src/interfaces/IInfraredBERAFeeReceivor.sol";
import {ValidatorTypes} from "src/core/libraries/ValidatorTypes.sol";
import {Errors} from "src/utils/Errors.sol";
import {DataTypes} from "src/utils/DataTypes.sol";
import {IBGTStaker} from "@berachain//pol/interfaces/IBGTStaker.sol";

contract WhitehatTest is Helper {
    function testharvestBaseFailsDueToBurningOfIBGT() public {
        // 1. Add a validator to the validator set
        vm.startPrank(infraredGovernance);
        ValidatorTypes.Validator memory validator_str = ValidatorTypes
            .Validator({
                pubkey: "0x1234567890abcdef",
                addr: address(validator)
            });
        ValidatorTypes.Validator[]
            memory validators = new ValidatorTypes.Validator[](1);
        validators[0] = validator_str;
        bytes[] memory pubkeys = new bytes[](1);
        pubkeys[0] = validator_str.pubkey;
        infrared.addValidators(validators);
        vm.stopPrank();

        // 2. Mint ibgt to some random address, such that total supply of ibgt is 100 ether
        vm.prank(address(infrared));
        ibgt.mint(address(12), 10000 ether);
        vm.startPrank(address(blockRewardController));
        // 3. Mint bgt to the Infrared, to simulate the rewards.
        bgt.mint(address(infrared), 10000 ether);
        vm.stopPrank();
        deal(address(bgt), 20000 ether);

        // Store initial balances
        uint256 receivorBalanceBefore = ibera.receivor().balance;

        //queue boosts
        vm.startPrank(keeper);
        uint128[] memory _amts = new uint128[](1);
        _amts[0] = 10000 ether;
        infrared.queueBoosts(pubkeys, _amts);
        //fastforward activation period 8191 blocks
        vm.roll(block.number + 8192);
        //activate boosts
        infrared.activateBoosts(pubkeys);
        vm.stopPrank();

        //send base rewards to Infrared
        vm.prank(address(blockRewardController));
        bgt.mint(address(infrared), 5000 ether);

        //burn ibgt
        vm.prank(address(12));
        ibgt.burn(1 wei);

        // 4. Call harvestBase to distribute the rewards
        vm.expectRevert(bytes4(keccak256("NotEnoughBalance()")));
        infrared.harvestBase();
    }
}
```

- Run the test.

Impact Explanation: Impact is Medium because.

- harvestBase will be DOSed until keeper calls queueDropBoost, then everyone has to wait for the dropBoostDelay of 8191 blocks to pass before the boosts get dropped.
- Functions that call harvestBase like addValidator, removeValidator, replaceValidator will be DOSed during this period.

Likelihood Explanation: Likelihood is Medium to High because the amount required to burn will usually be very tiny, so will be easily done by anyone. The reason why "the amount required to burn will usually be very tiny" is that protocol is incentivized to queue all queuable BGT in the contract as boosts, so that there will be more rewards available for vaults. The more BGT that gets queued, the less IBGT will be required to burn.

Recommendation:

- harvestBase should try to redeem :

```
    function harvestBase(
        address bgt,
        address ibgt,
        address ibera
    ) external returns (uint256 bgtAmt) {
        uint256 minted = IInfraredBGT(ibgt).totalSupply();
        uint256 bgtBalance = _getBGTBalance(bgt);
        // @dev should never happen but check in case
        if (bgtBalance < minted) revert Errors.UnderFlow();

++++    queuedBoost= IBerachainBGT(bgt).boosts(address(this)) + IBerachainBGT(bgt).queuedBoost(address(this));
++++    bgtAmt = bgtBalance - max(minted,queuedBoost);
        if (bgtAmt == 0) return 0;

        IBerachainBGT(bgt).redeem(IInfraredBERA(ibera).receivor(), bgtAmt);
    }
```

So as to prevent the revert.

3.2.40 The wrapped vault's claimRewards can be sandwiched to steal all/most of the reward token which equals the staking token

Severity: Medium Risk

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

Summary: No frontrunning is necessary in order to exploit this. Taking in consideration that the `claimRewards()` method on the WrappedVault contract is permissionless, a malicious user can sandwich a call to the `claimRewards()` with a deposit / withdraw call, in order to get the majority of the rewards in the case in which the reward token = staking token..

Due to the fact that a `claimRewards()` call would cause a "stepwise jump" in the reward valuation due to the accrual of assets in the contract, a malicious user can utilize flashloans (even interest-free ones) to steal the majority of the rewards..

Finding Description: There are a couple of pre-conditions that make this vulnerability plausible:

- Atomic deposit/withdrawals within the Wrapped Vault are allowed.
- `claimRewards()` isn't upkept atomically and can be permissionlessly called by anyone, this would allow malicious users to exploit any pending rewards to their gain.
- There are no minting taxes or reward indexes that would make this kind of attack unprofitable.

The wrapped vault's rewards are based on an index based on all rewards deposited to it. Since this will accumulate rewards in the Infrared Vault overtime, they need to be withdrawn by calling `claimRewards()` on the wrapped vault..

There's a condition in this function which omits the transfer of a claimed reward token, if said reward token equals the staking token (i.e. the asset of the vault). This is done so that the valuation of the vault share increases over time (assets > supply). The problem with this is, that it can easily be sandwiched by a malicious user utilizing a flashloan in order to steal all of the rewards from other users.

A detailed example of this is present in the Proof of Concept section of this report..

Considering that atomic deposits/withdrawals are allowed, frontrunning isn't needed as if enough of a reward accumulation is observed, the user can just call `claimRewards()` themselves (and sandwich it).

The above, coupled with lack of any minting/withdrawal tax present, allows for malicious users to sandwich any increase in the vault valuation from other users without staking for a prolonged time.

`claimRewards()`:

```
function claimRewards() external {
    // Claim rewards from the InfraredVault
    iVault.getReward();
    // Retrieve all reward tokens
    address[] memory _tokens = iVault.getAllRewardTokens();
    uint256 len = _tokens.length;
    // Loop through reward tokens and transfer them to the reward distributor
    for (uint256 i; i < len; ++i) {
        ERC20 _token = ERC20(_tokens[i]);
        // Skip if the reward token is the staking token
        if (_token == asset) continue;
        uint256 bal = _token.balanceOf(address(this));
        if (bal == 0) continue;
        (bool success, bytes memory data) = address(_token).call(
            abi.encodeWithSelector(
                ERC20.transfer.selector, rewardDistributor, bal
            )
        );
        if (success && (data.length == 0 || abi.decode(data, (bool)))) {
            emit RewardClaimed(address(_token), bal);
        } else {
            continue;
        }
    }
}
```

- From the above code excerpt we can see that the reward token balance is not sent to the distributor if the reward token = staking token, so that it can increase the "valuation" of the vault.

Refer to the below section for a sandwich scenario in which the malicious user steals 97% of the rewards at no extra cost (besides gas fees).

Impact Explanation: Malicious users are able to steal any significant jumps in the valuation of the wrapped vault achieved through a sandwich attack on the `claimRewards()`.

Likelihood Explanation: Anyone can exploit this, especially if flashloans are utilized.

Proof of Concept: Let's say that the current contract state is as follows:

- Total Supply: 1_000e18.
- Total Assets: 1_050e18 + 1e3.
- Let's say that the pending rewards (pending rewards which are in the same token as the staking one, e.g. iBGT) in the vault in which the user has deposited are 150e18;
- The malicious user utilizes a flashloan, and deposits 50_000e18 units of the staking token to the vault;
- shares = 50_000e18 * 1_000e18 / (1_050e18 + 1e3) = 47_619e18;
- The malicious user calls `claimRewards()`, which in return invokes the `getReward()` on the Infrared Vault; New contract state:
- Total Supply: 48_619e18;
- Total Assets: 51_200e18;
- Malicious user goes on to withdraw their shares in exchange of assets:
- 47_619e18 * 51_200e18 / 48_619e18 = 50_146e18;

With this, the malicious user has successfully sandwiched 97% of the increase in valuation of the vault (took 146e18 out of the 150e18 reward tokens).

Recommendation: There are multiple measurements which can be put in place in order to mitigate this:

- Don't allow atomic deposits/withdrawals (i.e. both performed in the same block).
- Introduce a small minting fee which would probably offset a lot of the possible attacks.
- Make sure that `claimRewards()` is called very frequently (ideally atomically through some automated flow at certain intervals) so that there aren't any stepwise jumps in the vault valuation / rewards.

3.2.41 Infrared.sol - function `queueNewCuttingBoard` will revert in Berachef contract due to `onlyOperator` check

Severity: Medium Risk

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

Description: Infrared.sol - keeper calls the function `queueNewCuttingBoard` to update the wights..

This function again calls the `queueNewRewardAllocation` function of Berachef contract.

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

if we look at the `queueNewRewardAllocation`.

```
function queueNewRewardAllocation(
    bytes calldata valPubkey,
    uint64 startBlock,
    Weight[] calldata weights
)
    external
    onlyOperator(valPubkey) -- check this
{
```

it expects the below - msg.sender should be the validator for the respective pubkey.

```
modifier onlyOperator(bytes calldata valPubkey) {
    if (msg.sender != beaconDepositContract.getOperator(valPubkey)) {
        NotOperator.selector.revertWith();
    }
    _;
}
```

but the msg.sender is `Infrared` contract. hence, the call will revert.

Impact: Call can not be completed successfully and weight can not be updated.

The operator themselves can update weight as per their wish.

Recommendation: Check whether the caller is Infrared contract instead of checking the operator address.