Raga Finance Security Review

Conducted by: Tejas Warambhe

10th June 2025

Contents

1. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.
Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

2. Introduction

A time-boxed security review of the Raga Finance protocol was performed, with a focus on the security aspects of the application's smart contracts implementation.

3. Security Assessment Summary

Review commit hash - [7bd995827d954db0be7c3db6939a0f492e47f7c7]

4. Scope of the audit:

```
src/vaults/RagaVaultRegistry.sol
src/vaults/RagaVault.sol
src/strategies/Berachain/BaseStrategy.sol
src/strategies/Berachain/Berapaw/interfaces/IBexVault.sol
src/strategies/Berachain/Berapaw/interfaces/IStrategy.sol
src/strategies/Berachain/Berapaw/interfaces/IBalancerQuer
y.sol
src/strategies/Berachain/Berapaw/interfaces/IRewardVaul.s
ol
src/strategies/Berachain/Berapaw/BerapawStrategy.sol
src/strategies/Berachain/Infrared/InfraredStrategy.sol
src/strategies/Berachain/Infrared/InfraredStableStrategy.
sol
src/interfaces/IRagaVault.sol
src/interfaces/IRagaVaultRegistry.sol
src/interfaces/IRouter.sol
```

5. About Raga Finance

Raga Finance is a vault-based DeFi protocol designed to simplify the Web3 investment experience across Layer 1 and Layer 2 ecosystems. By offering ERC-4626-compliant vaults, Raga enables both crypto-native and new users to explore, understand, and invest in on-chain strategies tailored to their individual risk profiles.

6. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

7. Table of Contents

I-1	Info	`Unused _swapIbgtForHoney function`	Dead code that should be removed
I-2	Info	`Documentation inconsistency in getTotalAssets()`	Function documentation doesn't match implementation

8. Summary by Severity

Severity	Count	Issues
High	3	Critical vulnerabilities affecting fund security and protocol functionality
Medium	4	Important issues affecting protocol operations and fee management
Low	3	Minor security and operational improvements
Informational	2	Code quality and documentation issues
Total	**12**	

[H-1] RagaVault::_withdraw reverts upon emergency withdraw

Description

The `RagaVault::_withdraw` is an internal call made while using the `redeem()` or `withdraw()` function of the `RagaVault` contract.

However, we can observe that the function uses difference of balances post withdrawal from the strategy:

```
    function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    )
        internal
        override
    {
        if (caller != owner) {
            _spendAllowance(owner, caller, shares);
        }
        _burn(owner, shares);
        uint256 beforeBal =
IERC20(asset()).balanceOf(address(this));                <<@ -- //
Tracks balance before
        IStrategy(_getVaultStorage().strategy).withdraw(assets);
<<@ -- // Withdrawal from startegy
        uint256 afterBal = IERC20(asset()).balanceOf(address(this));
<<@ -- // Tracks balance after
        SafeERC20.safeTransfer(IERC20(asset()), receiver, afterBal -
beforeBal);
        emit Withdraw(caller, receiver, owner, afterBal - beforeBal,
shares);
    }
```

However, in a case where the owner invokes `RagaVault::emergencyWithdraw` or `RagaVault::panic`, the tokens are withdrawn into the `RagaVault` contract. This contract lacks a way to retrieve tokens back to the users / owner.

Impact

1. Direct loss of funds can be observed here due to lack of token recovery options.

Proof of concept

Add the below test case inside `RagaVault.t.sol`:

```
    function testUserWithdrawRevertOnEmergency() public {
        uint256 amt = 1000 ether;
        token.mint(alice, 1000 ether);
        _depositFrom(alice, amt);

        // Move all funds into strategy so vault balance == 0
        assertEq(token.balanceOf(address(vault)), 0);

        uint256 withdrawAmt = 1000 ether;

        vault.pause();
        vault.emergencyWithdraw();

        vm.startPrank(alice);
        // Error: Insufficient Assets
        vm.expectRevert();
        vault.withdraw(withdrawAmt, alice, alice);
        vm.stopPrank();

        // Balance of Alice is unchanged, i.e, 0.
        assertEq(token.balanceOf(alice), 0 ether);
        // Vault receives the tokens
        assertEq(vault.totalAssets(), 1000 ether);
    }
```

Recommendations

It is intuitively better to allow users to withdraw funds irrespective of the state of the protocol being paused or not.

The share tracking is not maintained by the vault but rather dependent upon what the token balance the contract received, the mitigation might need a redesign of the withdrawal flow.

One of the solutions would be to allow proportional withdrawal via separate function once the emergency withdraw is triggered (which might be arguably the point where the vault will never be unpaused and a new vault would be deployed).

```
    /**
    * @notice Emergency proportional withdrawal for users when vault
is permanently paused
    * @dev This function allows users to withdraw their proportional
share of assets
    *      based on the current balance held by the vault contract
    * @param shares Amount of shares to burn for proportional
withdrawal
    * @return assets Amount of assets withdrawn
    */
    function emergencyProportionalWithdraw(uint256 shares)
        external
        whenPaused
        returns (uint256 assets)
    {
        if (shares == 0) revert ZeroAmount();
        if (shares > balanceOf(msg.sender)) revert("Insufficient
shares");

        // Get current asset balance in the vault (not including
strategy assets)
        uint256 vaultAssetBalance =
IERC20(asset()).balanceOf(address(this));

        if (vaultAssetBalance == 0) revert("No assets available for
withdrawal");

        // Calculate total supply of shares
        uint256 currentTotalSupply = totalSupply();

        if (currentTotalSupply == 0) revert("No shares in
circulation");

        // Calculate proportional assets based on vault balance only
        assets = (shares * vaultAssetBalance) / currentTotalSupply;

        if (assets == 0) revert ZeroAmount();

        // Burn the shares
        _burn(msg.sender, shares);

        // Transfer proportional assets to user
        IERC20(asset()).safeTransfer(msg.sender, assets);
```

```
        emit Withdraw(msg.sender, msg.sender, msg.sender, assets,
shares);
    }
```

Team Response

Acknowledged and Fixed

[H-2] Lack of scaling in `InfraredStrategy::withdraw` and `InfraredStableStrategy::withdraw` leads to loss of funds

Description

The `InfraredStrategy::withdraw` and `InfraredStableStrategy::withdraw` are designed to allow withdrawals via `RagaVault::_withdraw`.

However, when we have a closer look their implementation, we can find that the ratio calculation takes place without using a scaling factor:

```
    // File: InfraredStableStrategy.sol
    function withdraw(uint256 amount) external onlyVault
nonReentrant {
        StableStorage storage ss = _s();
        BaseStrategyStorage storage bs = _bs();

        uint256 lpPoolBalance = getLpPoolBalance();

        if (amount <= lpPoolBalance) {
            uint256 ratio = Math.mulDiv(amount, bs.MAX_BPS,
lpPoolBalance);                 <<@ -- // Lack of scaling
            uint256 ibgtBalance =
IRewardVault(ss.ibgtVault).balanceOf(address(this));
            uint256 ibgtAmountToWithdraw = Math.mulDiv(ibgtBalance,
ratio, bs.MAX_BPS);
            // Withdraw LP from wberaIbgtVault
            IRewardVault(ss.honeyByUsdVault).withdraw(amount);
            if (ibgtAmountToWithdraw > 0) {

IRewardVault(ss.ibgtVault).withdraw(ibgtAmountToWithdraw);
                amount += _exitAllPositions(ss);
            }
            // Transfer all the amount to the user
            want().transfer(bs.vault, amount);
        } else {
            revert InsufficientAssets();
        }
```

```
    }
```

```
    // File: InfraredStrategy.sol
    function withdraw(uint256 amount) external onlyVault
nonReentrant {
        InfraredStrategyStorage storage is_ = _is();
        BaseStrategyStorage storage bs_ = _bs();
        uint256 lpPoolBalance = getLpPoolBalance();

        if (amount <= lpPoolBalance) {
            uint256 ratio = Math.mulDiv(amount, bs_.MAX_BPS,
lpPoolBalance);                <<@ -- // Lack of scaling
            uint256 ibgtBalance =
IRewardVault(is_.ibgtVault).balanceOf(address(this));
            uint256 ibgtAmountToWithdraw = Math.mulDiv(ibgtBalance,
ratio, bs_.MAX_BPS);
            // Withdraw LP from wberaIbgtVault
            IRewardVault(is_.wBeraIbgtVault).withdraw(amount);

            if (ibgtAmountToWithdraw > 0) {

IRewardVault(is_.ibgtVault).withdraw(ibgtAmountToWithdraw);
                // convert the iBGT to LP
                amount += _buildWberaIbgtLp(is_);
            }

            want().transfer(bs_.vault, amount);
        } else {
            revert InsufficientAssets();
        }
    }
```

If the total `lpPoolBalance` is larger than the `amount * bs_.MAX_BPS`, the ratio would resolve to 0.

```
    lpPoolBalance: 20_000 ether
    amount: 1 ether
    MAX_BPS: 10_000
    Math.mulDiv is equivalent to floor(x * y / denominator)
    ratio = (amount * 10_000) / lpPoolBalance
          = (1e18 * 10_000) / (20_000 * 1e18)              <<@
Solidity rounds down
          = 0
```

Hence, users would lose excess rewards in such case as the block shown below would never run:

```
    if (ibgtAmountToWithdraw > 0) {
<<@ -- // Resolves to 0 in such cases.
        IRewardVault(is_.ibgtVault).withdraw(ibgtAmountToWithdraw);
        // convert the iBGT to LP
        amount += _buildWberaIbgtLp(is_);
    }
```

```
    if (ibgtAmountToWithdraw > 0) {
<<@ -- // Resolves to 0 in such cases.
        IRewardVault(ss.ibgtVault).withdraw(ibgtAmountToWithdraw);
        amount += _exitAllPositions(ss);
    }
```

Impact

1. Direct loss of funds to the users.
2. High ticket depositors would be enjoying higher rate of return.

Proof of Concept

Add the following test case inside the `InfraredStrategy.t.sol` file:

```
    function testRoundingError() public {
        // Setup: deposit and advance time
        (uint256 depositAmount,) = _depositAndAdvanceTime();

        // Record initial state
        uint256 initialStakedBalance =
IERC20(WBERA_IBGT_VAULT).balanceOf(address(strategy));
        uint256 aliceLpBalanceBefore =
IERC20(LP_TOKEN).balanceOf(alice);


        // Let time pass to accumulate rewards
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);

        strategy.harvest();

        // Execute partial withdrawal (50%)
        uint256 withdrawAmount = depositAmount / 2;
```

```
        vm.stopPrank();

        vm.startPrank(address(vault));
        // Balance of vault before
        uint vaultBeforeBal =
IERC20(LP_TOKEN).balanceOf(address(vault));
        // Trigger Withdraw
        strategy.withdraw(withdrawAmount);
        // Balance of vault after
        uint vaultAfterBal =
IERC20(LP_TOKEN).balanceOf(address(vault));

        uint difference = vaultAfterBal - vaultBeforeBal;

        // Proves that excess LP (rewards) were received
        assertGt(difference, withdrawAmount);

        // Simulate vault transferring tokens to user
        IERC20(LP_TOKEN).transfer(alice, difference);

        assertEq(IERC20(LP_TOKEN).balanceOf(address(vault)), 0);

        // Now, let's withdraw again after ofsetting 4 decimals
        vaultBeforeBal = IERC20(LP_TOKEN).balanceOf(address(vault));

        strategy.withdraw(withdrawAmount / 10000);

        vaultAfterBal = IERC20(LP_TOKEN).balanceOf(address(vault));

        difference = vaultAfterBal - vaultBeforeBal;

        // Proves that rounding down without scaling leads to loss
of funds for the user.
        assertEq(difference,  withdrawAmount / 10000);

        vm.stopPrank();

    }
```

Recommendations

It is advised to use a scaling factor or simply use the given values as they are already scaled individually (direct proportion).

```
function withdraw(uint256 amount) external onlyVault nonReentrant {
    InfraredStrategyStorage storage is_ = _is();
    BaseStrategyStorage storage bs_ = _bs();
    uint256 lpPoolBalance = getLpPoolBalance();

    if (amount <= lpPoolBalance) {
        uint256 ibgtBalance =
IRewardVault(is_.ibgtVault).balanceOf(address(this));

        // Calculate proportional iBGT amount to withdraw
        // Use direct proportion: ibgtAmountToWithdraw = (amount *
ibgtBalance) / lpPoolBalance
        uint256 ibgtAmountToWithdraw = 0;
        if (ibgtBalance > 0 && lpPoolBalance > 0) {
            ibgtAmountToWithdraw = Math.mulDiv(amount, ibgtBalance,
lpPoolBalance);
        }

        // Withdraw LP from wberaIbgtVault
        IRewardVault(is_.wBeraIbgtVault).withdraw(amount);

        if (ibgtAmountToWithdraw > 0) {

IRewardVault(is_.ibgtVault).withdraw(ibgtAmountToWithdraw);
            // convert the iBGT to LP
            amount += _buildWberaIbgtLp(is_);
        }

        want().transfer(bs_.vault, amount);
    } else {
        revert InsufficientAssets();
    }
}
```

Team Response

Acknowledged and Fixed

[H-3] Usage of `slot0` for `sqrtPriceX96` is highly prone to manipulation

Description

The InfraredStableStrategy::_calculateExpectedOutput and InfraredStrategy::_calculateExpectedOutput are used for facilitating swaps, where the expected amounts out is calculated as per the current `sqrtPriceX96` using the pool's `slot0`.

```
    // File: InfraredStableStrategy.sol
    function _calculateExpectedOutput(
        address tokenIn,
        address tokenOut,
        uint256 amountIn
    )
        internal
        view
        returns (uint256 expected)
    {
        // . . . Rest of the code . . .
        (uint160 sqrtPriceX96,,,,,) = pool.slot0();             <<@
        uint256 price = Math.mulDiv(uint256(sqrtPriceX96), 1e18, (1
<< 96)) ** 2;
        (address token0,) = tokenIn < tokenOut ? (tokenIn, tokenOut)
: (tokenOut, tokenIn);
        return tokenIn == token0 ? Math.mulDiv(amountIn, price,
1e36) : Math.mulDiv(amountIn, 1e36, price);
    }
```

```
    // File: InfraredStrategy.sol
    function _calculateExpectedOutput(
        address tokenIn,
        address tokenOut,
        uint256 amountIn
    )
        internal
        view
        returns (uint256)
    {
```

```
        // . . .Rest of the code . . .
        (uint160 sqrtP,,,,,,) = pool.slot0();
<<@
        uint256 price = Math.mulDiv(uint256(sqrtP) * uint256(sqrtP),
1e18, 1 << 192);
        // The below values are not price impact adjusted but we
expect very less price impact
        // due to small trade sizes + slippage
        // That's why in the minimum amount out we add a small
negative tolerance from this amount.
        return tokenIn == is_.wbera ? Math.mulDiv(amountIn, price,
1e18) : Math.mulDiv(amountIn, 1e18, price);
    }
```

However, this data is extremely prone to manipulation and can change multiple times in a single transaction.

An attack path for MEV bot / attacker would be to observe the public mempool, as soon as swap call is being observed, the attacker would immediately take a large flash loan in order to manipulate the pool's current price tick, which would sandwich a legitimate transaction in order to maximize profit.

P.S: The `_calculateMinimumOut` implementation does not help here as it calculates the minimum amount on the given expected value which is already manipulated.

Impact

1. Direct loss of funds.
2. Broken minimum amount out mechanism does not help in limiting potential loss.

Recommendation

It is recommended to use TWAP function instead of `slot0` to obtain the value of `sqrtPriceX96`.
TWAP (time-weighted average price) is a price averaging algorithm which would average out the current market price over a set period, hence more accurately representing value.

Team Response

Acknowledged

[M-1] RagaVault::panic reverts due to incorrect call order

Description

The `RagaVault::panic` is designed to emergency withdraw and pause the contract subsequently.

However, when we have a closer look at the function, we can observe that a `emergencyWithdraw()` call takes place before `_pause()` invocation:

```
    /// @notice Emergency withdraw everything from strategy to vault
and pause the vault.
    function panic() external onlyRegistryOwner whenNotPaused {
        emergencyWithdraw();    <<@ -- // 1
        _pause();               <<@ -- // 2
    }
```

And, the `emergencyWithdraw()` call observes a `whenPaused` modifier.

```
    /// @notice Emergency withdraw everything from strategy to vault
(paused state).
    function emergencyWithdraw() public onlyRegistryOwner whenPaused
{
        IStrategy(_getVaultStorage().strategy).withdrawAll();
    }
```

This would revert the `panic()` call.

Proof of concept

Add the below test case inside `RagaVault.t.sol`:

```
    function testPanicRevert() public {
        uint256 amt = 1000 ether;
        token.mint(alice, 1000 ether);

        _depositFrom(alice, amt);

        // error: ExpectedPause()
        vm.expectRevert();
        vault.panic();
    }
```

Recommendations

It is recommended to fix the call ordering:

```
    function panic() external onlyRegistryOwner whenNotPaused {
-         emergencyWithdraw();
+         _pause();
+         emergencyWithdraw();
-         _pause();
    }
```

Team Response

Acknowledged and Fixed

[M-2] Loss of fees upon emergency withdrawal

Description

The `BaseStrategy::collectFees` is used in order to send fees directly to the treasury.

The actual fee collection accounting takes place inside the `_calculateFee` function:

```
    function _calculateFee(uint256 grossProfit) internal returns
(uint256 fee) {
        if (grossProfit == 0) return 0;
        BaseStrategyStorage storage bs = _bs();
        IRagaVaultRegistry.Fee memory f =
IRagaVaultRegistry(IRagaVault(bs.vault).getRegistry()).getFees();
        if (f.performance == 0) return 0;
        fee = Math.mulDiv(grossProfit, f.performance, bs.MAX_BPS);
        bs.accumulatedFees += fee;      <<@ -- // Fee accumulation
    }
```

When the `BerapawStrategy::withdrawAll` or `InfraredStableStrategy::withdrawAll` or `InfraredStrategy::withdrawAll` call takes place, which would happen during emergency withdrawal, the entire balance is emptied out to the vault contract:

```
    // File: BerapawStrategy.sol
    function withdrawAll() external onlyVault {
       // . . . Rest of the code . . .

        uint256 totalOut = want().balanceOf(address(this));
<<@ -- // Uses entire contract balance
        s.totalAssets = 0;
        want().safeTransfer(bs.vault, totalOut);
<<@ -- // Transfers entire contract balance
    }
```

```
    // File: InfraredStableStrategy.sol
    function withdrawAll() external onlyVault nonReentrant {
       // . . . Rest of the code . . .
```

```
        // 5. Return funds to the vault
        uint256 bal = want().balanceOf(address(this));            <<@
-- // Uses entire contract balance
        want().safeTransfer(_bs().vault, bal);                    <<@
-- // Transfers entire contract balance
    }
```

```
    function withdrawAll() external onlyVault nonReentrant {
       // . . . Rest of the code . . .

        // -- 7. Transfer all available LP tokens to the vault
        uint256 lpAmount = want().balanceOf(address(this));
<<@ -- // Uses entire contract balance
        if (lpAmount > 0) {
            want().transfer(bs_.vault, lpAmount);
<<@ -- // Transfers entire contract balance
        }
    }
```

However, the [BaseStrategy::collectFees](#) uses `bs.accumulatedFees` value, which would still be greater than `0`. Hence, the fees would be simply lost here.

Even if the protocol tries to mitigate by calling the `collectFees` function first and then executing `RagaVault::panic`, an attacker would be able to frontrun the `RagaVault::panic` call with a transaction which would simply increase the `accumulatedFees`. The repercussions would be that even if the protocol resumes via `unpause`, the `collectFees` would fail due to excessive accounting.

Impact

1. Loss of funds to the protocol owner as the fees will be lost.
2. Denial of Service of `collectFees` function due to incorrect accounting.

Proof of concept

Add the following test case inside the `InfraredStrategy.t.sol` file:

```
    function testBrokenCollectFeeUponEmergency() public {
        // Setup fees
        uint256 currentFee = _setupPerformanceFees();

        // Execute harvest flow with time advancement
        (uint256 depositAmount, uint256 initialAssets) =
```

```
_depositAndAdvanceTime();
        uint256 finalAssets = _executeHarvestAndGetAssets();

        // Verify assets increased
        assertGt(finalAssets, initialAssets, "Strategy totalAssets
should increase after harvest");

        // Mocking withdrawAll
        vm.startPrank(address(vault));
        strategy.withdrawAll();
        vm.stopPrank();

        // Collect fees would duly revert
        vm.expectRevert();
        strategy.collectFees();

    }
```

Recommendations

It is recommended to add the `collectFees()` call inside the `withdrawAll()` across all strategies to ensure no residue remains.

Team Response

Acknowledged and Fixed

[M-3] Failed harvest call would DoS `InfraredStableStrategy::deposit` and `InfraredStrategy::deposit`

Description

The `InfraredStableStrategy::deposit` and `InfraredStrategy::deposit` harvests before making a deposit:

```
    // File: InfraredStableStrategy.sol
    function deposit() external onlyVault nonReentrant {
        StableStorage storage ss = _s();
        uint256 amt = want().balanceOf(address(this));
@>>     _harvest(); // Harvest any rewards before depositing
        want().approve(ss.honeyByUsdVault, amt);
        IRewardVault(ss.honeyByUsdVault).stake(amt);
    }
```

```
        // File: InfraredStrategy.sol
        function deposit() external onlyVault nonReentrant {
        InfraredStrategyStorage storage is_ = _is();
        uint256 amt = want().balanceOf(address(this));
@>>     _harvest(); // Harvest first to claim iBGT rewards and
convert them to LP tokens
        want().approve(is_.wBeraIbgtVault, amt);
        IRewardVault(is_.wBeraIbgtVault).stake(amt);
    }
```

Each harvest call internally makes a `_swap` call, which is simply a uniswap V3 swap.

However, this design choice has a flaw, a uniswap's swap call can fail under two circumstances:

1. Likely that dust amount swaps would fail due to insufficient liquidity or the uniswapV3 price tick moves to a range where insufficient liquidity is spotted.
2. Minimum amount out threshold is not met due to volatile market changes ("Too little received" error in uniswap).

Hence, a legitimate deposit would fail under such conditions.

Impact

1. Temporary DoS of `InfraredStableStrategy::deposit` and `InfraredStrategy::deposit` functions.

Recommendation

It is highly recommended to use a try-catch block for both functions:

```
    // File: InfraredStableStrategy.sol
    function deposit() external onlyVault nonReentrant {
        StableStorage storage ss = _s();
        uint256 amt = want().balanceOf(address(this));
-       _harvest();
+        try _harvest() {} catch {
+            // do nothing
+        }
        want().approve(ss.honeyByUsdVault, amt);
        IRewardVault(ss.honeyByUsdVault).stake(amt);
    }
```

```
        // File: InfraredStrategy.sol
        function deposit() external onlyVault nonReentrant {
        InfraredStrategyStorage storage is_ = _is();
        uint256 amt = want().balanceOf(address(this));
-       _harvest();
+        try _harvest() {} catch {
+            // do nothing
+        }
        want().approve(is_.wBeraIbgtVault, amt);
        IRewardVault(is_.wBeraIbgtVault).stake(amt);
    }
```

Team Response

Acknowledged

[M-4] Lack of harvest call during withdrawal leads to loss of funds for users

Description

The `InfraredStableStrategy::withdraw`, `InfraredStrategy::withdraw` and `BerapawStrategy::withdraw` calls are used in order to withdraw the liquidity to the vaults, which is eventually sent to the user.

However, none of these functions call the `harvest` call before withdrawing the liquidity, which would eventually lead to loss of funds for the user.

P.S: The `deposit` functions across all three contracts use the `harvest` call.

Impact

1. Direct loss of funds for the users due to lack of `harvest`.

Proof of concept

Add the below test case inside the `InfraredStrategy.t.sol` file:

```
    function testLackOfHarvestUponWithdrawal() public {
        // Setup: deposit and advance time
        (uint256 depositAmount,) = _depositAndAdvanceTime();

        // Record initial state
        uint256 initialStakedBalance =
IERC20(WBERA_IBGT_VAULT).balanceOf(address(strategy));
        uint256 aliceLpBalanceBefore =
IERC20(LP_TOKEN).balanceOf(alice);


        // Let time pass to accumulate rewards
        vm.warp(block.timestamp + 1 days);
        vm.roll(block.number + 100);
        vm.stopPrank();

         vm.startPrank(address(vault));

        // Execute withdrawal (50%)
        uint256 withdrawAmount = depositAmount / 2;
        // Balance of vault before
```

```
        uint vaultBeforeBal =
IERC20(LP_TOKEN).balanceOf(address(vault));
        strategy.withdraw(withdrawAmount);
        // Balance of vault after
        uint vaultAfterBal =
IERC20(LP_TOKEN).balanceOf(address(vault));

        uint differenceBeforeHarvest = vaultAfterBal -
vaultBeforeBal;

        // Simulate vault transferring tokens to user
        IERC20(LP_TOKEN).transfer(alice, differenceBeforeHarvest);

        // Harvest now
        strategy.harvest();

        vaultBeforeBal = IERC20(LP_TOKEN).balanceOf(address(vault));

        // Same amount withdraw, but after harvest
        strategy.withdraw(withdrawAmount);

        vaultAfterBal = IERC20(LP_TOKEN).balanceOf(address(vault));

        uint differenceAfterHarvest = vaultAfterBal -
vaultBeforeBal;

        // Proves that post harvest withdraw leads to a higher ROI
        assertGt(differenceAfterHarvest, differenceBeforeHarvest);

        vm.stopPrank();

    }
```

Recommendations

It is recommended to add a harvest call before actually withdrawing liquidity.

P.S:- This should be added inside the `RagaVault.sol` contract by overriding the redeem / withdraw function and not the strategy contract, this is critical.

```
    function redeem(uint256 shares, address receiver, address owner)
public override returns (uint256) {
        try IStrategy(_getVaultStorage().strategy).harvest() {}
catch {          <<@ -- // Try catch is important to ensure the
```

```
calls never fail due to uniswap / minimum amount out failure
            // Do nothing
        }
        uint256 maxShares = maxRedeem(owner);
        if (shares > maxShares) {
            revert ERC4626ExceededMaxRedeem(owner, shares,
maxShares);
        }

        uint256 assets = previewRedeem(shares);
        _withdraw(_msgSender(), receiver, owner, assets, shares);

        return assets;
    }
```

Similarly, for the withdraw call:

```
        function withdraw(uint256 assets, address receiver, address
owner) public override returns (uint256) {
        try IStrategy(_getVaultStorage().strategy).harvest() {}
catch {           <<@ -- // Try catch is important to ensure the
calls never fail due to uniswap / minimum amount out failure
            // Do nothing
        }
        uint256 maxAssets = maxWithdraw(owner);
        if (assets > maxAssets) {
            revert ERC4626ExceededMaxWithdraw(owner, assets,
maxAssets);
        }

        uint256 shares = previewWithdraw(assets);
        _withdraw(_msgSender(), receiver, owner, assets, shares);

        return shares;
    }
```

Team Response

Acknowledged

[L-1] Lack of ERC20 recovery function

Description

In DeFi, users / protocol unintentionally send tokens to incorrect contract and is a fairly common phenomenon, hence having a recover ERC20 function would help token recovery in such cases.
Currently, all the inscope contracts lack such a functionality.

Recommendation

It is recommended to add a `recoverERC20` function which does not touch the accounting of tokens used by the given contract, i.e, only allows withdrawal of ERC20 tokens other than tokens in interest of the contract.

Team Response

Acknowledged

[L-2] `BaseStrategy::setVault` can be frontran

Description

The `BaseStrategy::setVault` is used to set up the vault for the deployed strategy.

However, the access control modifier is public in nature allowing anyone to frontrun this call upon deployment.

```
    /// @notice Sets the vault address for this strategy only once.
    function setVault(address vault_) public virtual {      <<@ --
// Public in nature
        BaseStrategyStorage storage bs = _bs();
        if (bs.vault != address(0)) revert VaultAlreadySet();
        bs.vault = vault_;
```

```
        _afterVaultSet(vault_); // ← hook for strategy-specific
approvals or other stuff
        emit VaultSet(vault_);
    }
```

Recommendation

It is recommended to directly set the vault while strategy deployment takes place via the `_baseInit()` call.

Team Response

Acknowledged

[L-3] Anyone can call the collect fee function

Description

If ever the treasury needs to get updated due to whatever reason, this call can be frontran to withdraw fees to the old treasury.

```
    function collectFees() external {
        BaseStrategyStorage storage bs = _bs();
        uint256 fees = bs.accumulatedFees;
        if (fees == 0) return;
        bs.accumulatedFees = 0;
        address treasury =
IRagaVaultRegistry(IRagaVault(bs.vault).getRegistry()).getTreasury()
;
        want().safeTransfer(treasury, fees);
        emit FeeCollected(fees);
    }
```

Recommendation

It is recommended to only allow the admins / whitelisted addresses to call this function.

Team Response

Acknowledged and Fixed

[I-1] Unused `InfraredStableStrategy::_swapIbgtForHoney` function

Description

The `InfraredStableStrategy::_swapIbgtForHoney` is not being used anywhere, and can be considered as dead code:

```
    function _swapIbgtForHoney(uint256 amountIn) internal {
<<@ -- // Unused function
        // . . . Rest of the code . . .
    }
```

Recommendation

It is recommended to remove the function altogether.

Team Response

Acknowledged and Fixed

[I-2] Documentation Inconsistency in `InfraredStableStrategy::getTotalAssets()`

Description

The `InfraredStableStrategy::getTotalAssets()` function is used in order to calculate the `totalAssets` in the `RagaVault` contract.

The code comment suggests that the total assets of the strategy refers to asset staked in the vault + iBGT staked converted to LP.
However, the function returns only the current value of LP staked in the vault:

```
    /// @notice Returns the LP asset staked in the vault plus the
staked iBGT converted to LP
    function getTotalAssets() public view returns (uint256) {
<<@
        return getLpPoolBalance();
    }
```

Recommendation

It is recommended to fix the documentation.

Team Response

Acknowledged and Fixed