SHERLOCK

Security Review For Beraborrow

Private Audit Contest Prepared For:	**Beraborrow**
Lead Security Expert:	**0x73696d616f**
Date Audited:	**January 13 - January 31, 2025**
Final Commit:	**be7caf2**

Introduction

Beraborrow unlocks instant liquidity against Berachain assets through the first PoL powered stablecoin, Nectar ($NECT). Built with simplicity and flexibility at its core, Beraborrow is designed to maximise opportunities for users without forcing them to sacrifice yield.

Scope

Repository: Beraborrowofficial/blockend

Audited Commit: 3f331ebb0c371d6afdb8afdf9cb08d47bbfe20d3

Final Commit: be7caf251eb4a695a35fd793a061e2336b04eae9

Files:

- src/core/BeraborrowCore.sol
- src/core/BorrowerOperations.sol
- src/core/DebtToken.sol
- src/core/DenManager.sol
- src/core/Factory.sol
- src/core/GasPool.sol
- src/core/LiquidStabilityPool.sol
- src/core/LiquidationManager.sol
- src/core/MetaBeraborrowCore.sol
- src/core/PriceFeed.sol
- src/core/SortedDens.sol
- src/core/UpgradeableProxy.sol
- src/core/boyco/BoycoVault.sol
- src/core/boyco/PSMBond.sol
- src/core/boyco/PermissionedDenManager.sol
- src/core/helpers/DenManagerGetters.sol
- src/core/helpers/LiquidStabilityPoolGetters.sol
- src/core/helpers/MultiCollateralHintHelpers.sol
- src/core/helpers/MultiDenGetter.sol
- src/core/spotOracles/KodiakIslandFeed.sol
- src/core/vaults/BaseCollateralVault.sol

- src/core/vaults/InfraredCollateralVault.sol
- src/core/vaults/KodiakIslandVault.sol
- src/core/vaults/UsdcVault.sol
- src/core/vaults/bHONEYVault.sol
- src/core/vaults/iBGTVault.sol
- src/dao/PollenToken.sol
- src/dependencies/BeraborrowBase.sol
- src/dependencies/BeraborrowMath.sol
- src/dependencies/BeraborrowOwnable.sol
- src/dependencies/DelegatedOps.sol
- src/dependencies/SystemStart.sol
- src/libraries/EmissionsLib.sol
- src/libraries/FeeLib.sol
- src/libraries/LSPStorageLib.sol
- src/libraries/PriceLib.sol
- src/libraries/ReentrancyGuardLib.sol
- src/libraries/TokenValidationLib.sol
- src/libraries/UtilsLib.sol
- src/periphery/ChronicleWrapper.sol
- src/periphery/CollVaultRouter.sol
- src/periphery/LSPOracle.sol
- src/periphery/LSPRouter.sol
- src/periphery/ValidatorPool.sol

Final Commit Hash

be7caf251eb4a695a35fd793a061e2336b04eae9

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
0	15

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x73696d616f
0xeix
Afriaudit
Flashloan44
TessKimy
araj
bbl4de
bughuntoor
newspacexyz
pseudoArtist
vinica_boy

Issue M-1: Setting `peripheryFlashLoanFee` does not affect the effective fee value due to incorrect `getPeripheryFlashLoanFee()` implementation

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/1

The protocol has acknowledged this issue.

Found by

0x73696d616f, bbl4de

Summary

In `DebtToken` when flashloan is initiated the `_flashFee()` function is used to determine the effective fee:

```
function _flashFee(uint256 amount) internal view returns (uint256) {
    uint effectiveFee = _beraborrowCore.getPeripheryFlashLoanFee(msg.sender);

    return (amount * effectiveFee) / 1e4;
}
```

However, because the `getPeripheryFlashLoanFee()` function is called on the `BeraborrowCore` contract, the `msg.sender` in `MetaBeraborrowCore`, where the `peripheryFlashLoanFee` is set, will not pass the check:

```
if (msg.sender == nect) {
    if (info.existsForNect) {
        return info.nectFee;
    }
}
```

always returning the `DEFAULT_FLASH_LOAN_FEE`.

Root Cause

`getPeripheryFlashLoanFee` incorrectly checks the `msg.sender` value and the `DebtToken` contract calls `BeraborrowCore` instead of `MetaBeraborrowCore`.
https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/MetaBeraborrowCore.sol#L178-L182

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

No response

Impact

It is impossible to implement a flashloan fee reduction/increase for any periphery contract.

PoC

Place the following test in `DebtToken.t.sol`:

```
function test_getPeripheryFlashLoanFee() public {
    uint16 newFee = 1; // 0.05% --> 0.01%
    address periphery = makeAddr("periphery");

    vm.prank(owner);
    metaBeraborrowCore.setPeripheryFlashLoanFee(periphery, newFee, true);

    vm.prank(periphery);
    uint fee = nectarToken.flashFee(address(nectarToken), 1e4);
    assertEq(fee , 5);

}
```

Mitigation

Make the following change in `getPeripheryFlashLoanFee()`:

```
-       if (msg.sender == nect) {
            if (info.existsForNect) {
                return info.nectFee;
            }
-       }
```

To support calling that method through `BeraborrowCore` first or `MetaBeraborrowCore` directly.

Issue M-2: During Recovery Mode Liquidation, the TCR is being computed incorrectly

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/2

Found by

Flashloan44, TessKimy

Summary

During liquidation in recovery mode, protocol implements separate tracking of Total Collateral Ratio (TCR) (see line 350 and 526 below) which is incorrectly calculated because the **entireSystemColl** value has a missing subtraction of collateral gas compensation that lead to inaccuracy of computation. This will inflate the value of TCR and affects both liquidateDens and batchLiquidateDens

```
File: LiquidationManager.sol
337:             entireSystemColl -= totals.totalCollToSendToSP *
↪  denManagerValues.price; //@audit should subtract also the col gas compensation

~ continue
350:                         uint256 TCR =
↪  BeraborrowMath._computeCR(entireSystemColl, entireSystemDebt); //@audit TCR
↪  here is inflated

~ continue
364:                 entireSystemColl -=
365:                     (singleLiquidation.collToSendToSP +
↪  singleLiquidation.collSurplus) * ////@audit should subtract also the col gas
↪  compensation
366:                     denManagerValues.price;
```

```
File: LiquidationManager.sol
503:             entireSystemColl -= totals.totalCollToSendToSP *
↪  denManagerValues.price; //@audit should subtract also the col gas compensation
~ continue
526:                         uint256 TCR =
↪  BeraborrowMath._computeCR(entireSystemColl, entireSystemDebt); //@audit TCR
↪  here is inflated
~continue
541:                 entireSystemColl -=
542:                     (singleLiquidation.collToSendToSP +
↪  singleLiquidation.collSurplus) * //@audit should subtract also the col gas
↪  compensation
```

```
543:                     denManagerValues.price;
```

entireSystemColl is supposed to track the total system collateral during liquidations. While it correctly deducts the collateral sent to the Stability Pool and the surplus collateral amount, it fails to account for the collateral gas compensation.

The collateral gas compensation is necessary to deduct so the system can compute the real value of TCR. Remember that collateral gas compensation is being sent as reward for liquidators, snect gauge and validator pool, in which being deducted from actual system collateral. If we don't deduct it, the TCR is inflated, that will affect the liquidation of dens, like for example , that a den should be liquidated but due to inflated TCR , it will become higher than CCR, therefore no liquidation.

Root Cause

The root cause can be trace in a situation where the system's TCR falls below the Critical Collateralization Ratio (CCR), triggering recovery mode in which proper liquidation will apply.

The bug arises here because the TCR is inflated as collateral gas compensation is not deducted from the system's collateral. The issue is present in both liquidateDens and batchLiquidateDens

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

See POC section for vulnerability path.

Impact

TCR is artificially inflated, it means calculated TCR during liquidation is higher than actual. It means that Dens higher than actual TCR will be wrongly liquidated. Another case is that liquidation should happen but won't because the TCR is inflated in which it becomes higher than CCR. Therefore no liquidation happened.

PoC

Please follow first the instruction here in the secret gist link before running the test.

1. Insert this test under this file blockend/test/core/LiquidationManager.t.sol

```
function test_liquidateDensTCRDiff() external {

        _openDenA(depositor); // Open DenA with 185% ICR
        _openDenB(random);// Open DenB with 250% ICR
        _openDenC(depositor2);// Open DenC 221% ICR

        // After 12 seconds
        vm.warp(block.timestamp + 12 seconds);
        // Enable recovery mode, price drops to 60% of initial price
        _mockPriceFeed(6e17, block.timestamp, 2);

        // LiquidateDens
        vm.prank(addrs.factory);
        vm.recordLogs();
        liquidationManager.liquidateDens(denManager, 2, MCR, addrs.factory);
        VmSafe.Log[] memory logss = vm.getRecordedLogs();


        // Get final system TCR
         uint256 finalTCR = borrowerOperations.getTCR();
        (uint256 postColl, uint256 postDebt) =
↪   borrowerOperations.getGlobalSystemBalances();
        console.log("\nAfter-liquidation:");
        console.log("Final Collateral:", postColl);
        console.log("Final Debt:", postDebt);

        // Parse TCR events and verify difference
    for (uint i = 0; i < logss.length; i++) {
        if (logss[i].topics[0] ==
↪   keccak256("TCRCalculated(uint256,uint256,uint256)")) {
            (uint256 calculatedTCR, uint256 usedColl, uint256 usedDebt) =
↪   abi.decode(logss[i].data, (uint256,uint256,uint256));
            console.log("Used Collateral in calculation:", usedColl);
            console.log("Used Debt in calculation:", usedDebt);
            console.log("Calculated TCR:", calculatedTCR);
            console.log("Final TCR:", finalTCR);
            console.log("TCR Difference:", calculatedTCR - finalTCR);
        }
    }
    }
```

2. Run the test forge test -vv --match-contract LiquidationManagerTest --match-test test_liquidateDensTCRDiff

3. Result of the test: As you can see below there is difference between Actual (Final) TCR and Calculated TCR. This represents the unaccounted subtraction of collateral compensated gas.

```
After-liquidation:
  Final Collateral: 28752000000000000000000000000000000
  Final Debt: 2237130008512671232
  Used Collateral in calculation: 28800000000000000000000000000000000000
  Used Debt in calculation: 2237130008512671232
  Calculated TCR: 1287363715582507915
  Final TCR: 1285218109389870402
  TCR Difference: 2145606192637513

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 188.06s (5.35s CPU
↪  time)
```

Mitigation

Deduct collateral gas compensation when maintaining entireSystemColl.

```
-          entireSystemColl -= totals.totalCollToSendToSP * denManagerValues.price;
+         entireSystemColl -= (totals.totalCollGasCompensation +
↪  totals.totalCollToSendToSP) * denManagerValues.price;
```

```
-        entireSystemColl -= (singleLiquidation.collToSendToSP +
↪  singleLiquidation.collSurplus) * denManagerValues.price;
+       entireSystemColl -=(singleLiquidation.collToSendToSP +
↪  singleLiquidation.collSurplus + singleLiquidation.collGasCompensation) *
↪  denManagerValues.price;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/170

Issue M-3: Redemption fees can be manipulated during Recovery mode.

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/9

The protocol has acknowledged this issue.

Found by

Flashloan44

Summary

During Recovery mode, the borrowing fees are waived (not charged) when opening a den , this can be taken advantage by a malicious borrower to manipulate the redemption fees.

All he can do is to open a den with large debt during recovery mode with no borrowing fees and this will directly affect the redemption rate because redemption rate computation depend on total borrowed amount of the system and collateral being redeemed. The lower the collateral being redeemed compared to total debt of the system , the lower the redemption fees that will be charge for that collateral.

Look at the function below in line 507, total debt supply can be inflated in which will lower the redemption rate at the end. This is true specially if the collateral being drawn or redeemed is much smaller compared to total debt of the system.

```
File: DenManager.sol
498:     function _updateBaseRateFromRedemption(
499:         uint256 _collateralDrawn,
500:         uint256 _price,
501:         uint256 _totalDebtSupply
502:     ) internal returns (uint256) {
503:         uint256 decayedBaseRate = _calcDecayedBaseRate();
504:
505:         /* Convert the drawn collateral back to debt at face value rate (1
 ↪  debt:1 USD), in order to get
506:          * the fraction of total supply that was redeemed at face value. */
507:         uint256 redeemedDebtFraction = (_collateralDrawn * _price) /
 ↪  _totalDebtSupply; //@audit _totalDebtSupply can be manipulated that can affect
 ↪  the redemption rate at the end
508:
509:         uint256 newBaseRate = decayedBaseRate + (redeemedDebtFraction / BETA);
510:         newBaseRate = BeraborrowMath._min(newBaseRate, DECIMAL_PRECISION); //
 ↪  cap baseRate at a maximum of 100%
511:
512:         // Update the baseRate state variable
```

```
513:         baseRate = newBaseRate;
514:         emit BaseRateUpdated(newBaseRate);
515:
516:         _updateLastFeeOpTime();
517:
518:         return newBaseRate;
519:     }
520:
```

Root Cause

The root cause is allowing malicious actor an opportunity to manipulate redemption rate through inflating the total debt of the system through opening a den with large debt. There should be somehow way to discourage them for doing this and the solution will be applying borrowing fees during Recovery mode period.

Internal Pre-conditions

The system is in recovery mode.

External Pre-conditions

No response

Attack Path

This can be the scenario.

1. An attacker has a remaining open Den in the protocol. Let's call it Den A with 1e18 debt.
2. The protocol enters into Recovery Mode.
3. The attacker knowing the vulnerability, open another Den with very large debt of 1e20. Let's call it Den B
4. Since the total debt of the system increases significantly, this will affect redemption fee rate for redeeming Den A and becomes lower.
5. Attacker take advantage of the situation and redeemed Den A with lower redemption fee.
6. Attacker closes Den B as if nothing happened and system back into original state.

Impact

Manipulation of redemption fees to lower it. Loss of funds for the protocol

PoC

Here are the coded PoCs to show on how the redemption rate becomes lower between the two scenarios: **Note**: Please follow the instruction in this gist link before running the PoCs

1. Scenario 1 shows a normal scenario wherein no manipulation happened. No manipulation means no opening of large den before redemption of collateral. The rate increased by 4.5e16 or 4.5%, from 0.5% to 5% equal to max redemption rate of 5%.

```
// Scenario 1 wherein no manipulation happened.
    function test_redeemRatesDiffA() external {

        // Open Den with 1e18 debt
        _openDen1(depositor);

        // After 12 seconds
        vm.warp(block.timestamp + 12 seconds);

        // Enable recovery mode, price drops to 80% of initial price, 80% because
↪   redemption requires TCR >= MCR
        _mockPriceFeed(8e17, block.timestamp, 2);

        // redemption rate before redeem
        uint256 redemptionRatebefore = denManager.getRedemptionRate();

        // depositor perform redemption
        vm.startPrank(depositor);

       // to record the redemption rate during redeem
        vm.recordLogs();
         denManager.redeemCollateral({
            _debtAmount: 1005000003824200913, // to fully redeem den 1
            _firstRedemptionHint: address(0),
            _upperPartialRedemptionHint: address(0),
            _lowerPartialRedemptionHint: address(0),
            _partialRedemptionHintNICR: uint256(0),
            _maxIterations: uint256(0),
            _maxFeePercentage: 5e16 // should match with max redemption rate of 5%
             });
        VmSafe.Log[] memory logss = vm.getRecordedLogs();

        // Parse TCR events and verify difference
    for (uint i = 0; i < logss.length; i++) {
        if (logss[i].topics[0] == keccak256("RedemptionRate(uint256)")) {
            (uint256 redemptionRateafter) = abi.decode(logss[i].data, (uint256));
            console.log("\nRate Increase Summary:");
            console.log("Redemption Rate before redeem:", redemptionRatebefore);
            console.log("Redemption Rate during redeem:", redemptionRateafter);
```

```
            console.log("Rate Increased by:",  redemptionRateafter -
↪  redemptionRatebefore );
        }
    }
}
```

Command to run: forge test -vv --match-contract DenManagerTest --match-test test_redeemRatesDiffA

```
Rate Increase Summary:
  Redemption Rate before redeem: 5000000000000000
  Redemption Rate during redeem: 50000000000000000
  Rate Increased by: 45000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 172.69s (2.15s CPU
↪  time)
```

2. Scenario 2 shows the manipulation wherein there is a opening of large den before redemption of collateral. The rate increased only by 4.74e15 or 0.474%, from 0.5% to 0.974%. This is much lower than previous scenario 1.

```
//Scenario 2 wherein manipulation happened. The manipulation here is opening of
↪  large den before redemption
    function test_redeemRatesDiffB() external {

        // Open den with debt of 1e18 by malicious depositor
        _openDen1(depositor);

        // After 12 seconds
        vm.warp(block.timestamp + 12 seconds);

        // Enable recovery mode, price drops to 65% of initial price
        _mockPriceFeed(6.5e17, block.timestamp, 2);

        // redemption fee rate before manipulation
        uint256 redemptionRatebefore = denManager.getRedemptionRate();

        // Open large Den with 1.05e20 debt, manipulation attack through another
↪  account by depositor
        _openDen2(random);

        // depositor perform redemption
        vm.startPrank(depositor);

         // to record the redemption rate during redeem
        vm.recordLogs();
         denManager.redeemCollateral({
            _debtAmount: 1005000003824200913, // to fully redeem den 1
            _firstRedemptionHint: address(0),
```

```
            _upperPartialRedemptionHint: address(0),
            _lowerPartialRedemptionHint: address(0),
            _partialRedemptionHintNICR: uint256(0),
            _maxIterations: uint256(0),
            _maxFeePercentage: 5e16 // should match with max redemption rate of 5%
        });
        VmSafe.Log[] memory logss = vm.getRecordedLogs();

        // Parse TCR events and verify difference
    for (uint i = 0; i < logss.length; i++) {
        if (logss[i].topics[0] == keccak256("RedemptionRate(uint256)")) {
            (uint256 redemptionRateafter) = abi.decode(logss[i].data, (uint256));
            console.log("\nRate Increase Summary:");
            console.log("Redemption Rate before redeem:", redemptionRatebefore);
            console.log("Redemption Rate during redeem:", redemptionRateafter);
            console.log("Rate Increased by:",  redemptionRateafter -
↪   redemptionRatebefore );
        }
    }
 }
```

Command to run: forge test -vv --match-contract DenManagerTest --match-test test_redeemRatesDiffB

```
Rate Increase Summary:
  Redemption Rate before redeem: 5000000000000000
  Redemption Rate during redeem: 9740342454544336
  Rate Increased by: 4740342454544336
```

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 163.22s (2.95s CPU time)

3. Compare the two scenarios and the scenario 2 will show much lower increase of redemption rate compare to scenario 1. This is a proof that you can lower the redemption rate by simply inflating the total debt of the system through opening a large den.

Description	**(A)** Scenario 1	**(B)** Scenario 2	**(A - B = C)** Increase Difference
Rate Increased by	45000000000000000	4740342454544336	40259657545455664

Let's compute also how much in percentage did the redemption rate change from Scenario 1 to Scenario 2.

Increase difference of 40259657545455664 divided by Scenario 1 rate increase of 45000000000000000 is equal to 89.47%. So meaning there is 89.47% reduction in redemption rate after attacker conducts manipulation through opening a den with large debt. The 89.47% represent the loss of the protocol.

Mitigation

Apply lower borrowing fee during recovery mode. This fee should be lower than the fee imposed during normal mode to make encouragement to still open den during recovery mode. This is the way to incentivize the borrower to open den but discourage malicious attack of manipulating redemption fees. Making the process more balanced.

Issue M-4: Anyone can avoid receiving bad debt distribution by closing his Den right before liquidation

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/10

The protocol has acknowledged this issue.

Found by

Flashloan44, TessKimy

Summary

The protocol implements redistribution of a liquidated Den's bad debt during its liquidation. This bad debt will be shared to the remaining dens of the Den Manager.

However, any user can prevent receiving bad debt if they will close their dens right before liquidation. This could result race condition and every one want to close their dens. If there is only 1 den remained open, he will receive all bad debts which will further increase his debt. This can have a devastating impact because either he will be forced to pay that debt to recover the collateral or be liquidated.

Root Cause

There is no preventive measure applied by the protocol in this kind of situation. The current protocol design somehow allows it to happen. Closing of den can be freely executed anytime the user want in which can't be help during this kind of situation.

Internal Pre-conditions

1. Den with large bad debt is being liquidated.

External Pre-conditions

No response

Attack Path

This can be the scenario.

1. There are 3 remaining dens in the den manager. Den A, B and C.
2. Den A and B each hold the same amount of collateral and debt and in healthy state while C is in near liquidation risk.

3. Suddenly, collateral price drops and Den C accumulated bad debt.
4. Den B owner knew about vulnerability and immediately closed his den right before liquidation.
5. At this point, only 1 den remaining which is Den A, in which it receives all bad debt distribution from Den C liquidation.

Impact

This can impact the remaining Dens that will unfairly receive the bad debt. This will increase their debt and forced to pay for it to recover their collateral or avoid liquidation.

PoC

See attack path

Mitigation

Apply a mechanism in which there is a time delay of closing den during imminent liquidation of bad debt. Making sure the debt distribution is fair for everyone.

Issue M-5: Wrong slippage check in LSPRouter::withdraw()

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/18

Found by

vinica_boy

Summary

Usually, slippage related to ERC4626 vaults is either ensuring that users does not get less amount of assets when redeeming or does not burn more shares when withdrawing. In the `LSPRouter::withdraw()` implementation user can only provide `minSharesWithdrawn` which is not protects him from burning more shares than expected. The opposite check in `redeem()` is correctly implemented to check if actually withdrawn assets are at least what user expects.

Root Cause

The following snippet is part of `LSPRouter::withdraw()`:

```
shares = lsp.withdraw(params.assets, arr.receiver, msg.sender);
require(shares >= params.minSharesWithdrawn, "LSPRouter: assetsWithdrawn <
↪  minAssetsWithdrawn");
```

Slippage check in `withdraw()`: https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/periphery/LSPRouter.sol#L249

Another thing we have to take into consideration is the share:assets ratio in LSP. Usually, it should always be rising as more and more liquidations happen in LSP accrue funds from those liquidations. But there is a scenario in which the debt to offset from LSP is higher than the collateral to add leading to a decreasing ratio leading to more burned shares for the same amount.

In LiquidStabilityPool:

```
// Unlikely case in which LM offsets more debt value than collateral
uint collSurplusAmount;
if (_collToAdd > debtInCollateralAmount) {
    collSurplusAmount = _collToAdd - debtInCollateralAmount;
}
```

Apart from the potential ratio decrease, we have to take into consideration the prices of the underlying collateral (LSP consists of NECT + other priceable assets). A price change before the user transaction will affect the number of shares burned as well.

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

Users do not have a way to protect themselves from unexpected slippage.

Impact

Loss of funds for users.

N/A

Mitigation

Consider changing the check to:

```
require(shares <= params.maxSharesWithdrawn, "LSPRouter: assetsWithdrawn <
 ↪  minAssetsWithdrawn");
```

and the corresponding parameter.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/173

Issue M-6: Incorrect debt calculation for BrimeDen in DenManager leads to incorrect ICR calculation.

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/27

Found by

araj, newspacexyz, pseudoArtist

Summary

When calculate ICR in DenManager, it will calculate the borrower's entire debt with accrued den interest. But BrimeDen does not pay any interest. This results in the protocol will cause BrimeDen's wrong ICR calculation, it will result in incorrect liquidation of BrimeDen in the LiquidationManager.

Vulnerability Detail

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/DenManager.sol#L443

Whenever calculate ICR in DenManager instance, `getCurrentICR` will be called.

```
function getCurrentICR(address _borrower, uint256 _price) public view returns
↪  (uint256) {
@>> (uint256 currentCollateral, uint256 currentDebt) = getDenCollAndDebt(_borrower);

    uint256 ICR = BeraborrowMath._computeCR(currentCollateral, currentDebt, _price);
    return ICR;
}
```

In `getDenCollAndDebt` in DenManager.sol, it will call `getEntireDebtAndColl` function.

But in `getEntireDebtAndColl` function, it does not check borrower is brimeDen.

```
function getEntireDebtAndColl(
    address _borrower
) public view returns (uint256 debt, uint256 coll, uint256 pendingDebtReward,
↪  uint256 pendingCollateralReward) {
    Den storage t = Dens[_borrower];
    debt = t.debt;
    coll = t.coll;

    (pendingCollateralReward, pendingDebtReward) =
↪  getPendingCollAndDebtRewards(_borrower);
```

```
    // Accrued den interest for correct liquidation values. This assumes the index
↪   to be updated.
    uint256 denInterestIndex = t.activeInterestIndex;
    if (denInterestIndex > 0) {
        (uint256 currentIndex, ) = _calculateInterestIndex();
@>>     debt = (debt * currentIndex) / denInterestIndex;
    }

    debt = debt + pendingDebtReward;
    coll = coll + pendingCollateralReward;
}
```

But BrimeDen does not pay any interest. This miscalculation will cause BrimeDen's ICR value as lower. (ICR = coll / debt)

Impact

When BrimeDen holds debt within a DenManager, an incorrect calculation of BrimeDen's ICR will gradually decrease over time. This issue can lead to an unexpected liquidation in the LiquidationManager.

```
function liquidateDens(IDenManager denManager, uint256 maxDensToLiquidate, uint256
↪   maxICR, address liquidator) public {
    ...
@>> uint ICR = denManager.getCurrentICR(account, denManagerValues.price);
    uint applicableMCR = _getApplicableMCR(ICR, account, denManagerValues);
    if (ICR > maxICR) {
        ...
    }
@>> if (ICR <= _LSP_CR_LIMIT) {
        singleLiquidation = _liquidateWithoutSP(denManager, account);
        _applyLiquidationValuesToTotals(totals, singleLiquidation);
@>> } else if (ICR < applicableMCR) {
        singleLiquidation = _liquidateNormalMode(
            denManager,
            account,
            debtInStabPool,
            denManagerValues.sunsetting
        );
        debtInStabPool -= singleLiquidation.debtToOffset;
        _applyLiquidationValuesToTotals(totals, singleLiquidation);
    } else break; // break if the loop reaches a Den with ICR >= MCR
}
```

Tool used

Manual Review

Recommendation

```
function getEntireDebtAndColl(
    address _borrower
) public view returns (uint256 debt, uint256 coll, uint256 pendingDebtReward,
↪   uint256 pendingCollateralReward) {
    ...
    // Accrued den interest for correct liquidation values. This assumes the index
↪   to be updated.
    uint256 denInterestIndex = t.activeInterestIndex;
--  if (denInterestIndex > 0) {
++  if (denInterestIndex > 0 && _borrower != brimeDen) {
        (uint256 currentIndex, ) = _calculateInterestIndex();
        debt = (debt * currentIndex) / denInterestIndex; // @audit what's this
    }

    debt = debt + pendingDebtReward;
    coll = coll + pendingCollateralReward;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/169

Issue M-7: InfraredCollateralVault is incompatible with assets which has less than 18 decimals

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/28

Found by

vinica_boy

Summary

`InfraredCollateralVault` always scales the `totalAssets()` result up to 18 decimals. I believe it was added as a fix for issue M-5 for the first Sherlock audit (November 20 - December 27). While this is fixing the problem with `totalAssets()` accounting, it introduces another problem when doing shares conversion.

Root Cause

`InfraredCollateralVault` overrides the ERC4626 `totalAssets()` implementation to include reward tokens value and scale the result to 18 decimals. This makes huge inconsistency between the first minted shares and all subsequently minted shares.

Taking a look into the `_convertToShares()` method of inherited OZ ERC4626 implementation:

```
function _convertToShares(uint256 assets, Math.Rounding rounding) internal view
↪  virtual returns (uint256) {
    return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(), totalAssets() +
↪  1, rounding);
}
```

and we have for `_decimalsOffset()`:

```
function _decimalsOffset() internal view override virtual returns (uint8) {
    return 18 - assetDecimals();
}
```

Initially, `totalAssets()` and `totalSupply()` would be 0 as no assets have been deposited and donation is not possible. So, lets say that the first depositor want to add 100e6 (asset is with 6 decimals, decimalsOffset is 12) of the underlying assets, which would result in `(100e6 * (0+1e12))/1 = 100e18 shares`. This minting is correct and shares are correctly scaled up to 18 decimals because of the decimalsOffset.

Next depositor also want to deposit 100e6 assets. Going through the formula for shares conversion we have `(100e6 * (100e18 + 1e12))/100e18`. Note that actual asset balance is 100e6,

but `totalAssets()` return is scaled to 18 decimals. Shares for the second mint are ~ `10000e24/100e18 = 100e6` which is significantly lower than the initially minted 100e18 shares.

`InfraredCollateralVault::totalAssets()`: https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/InfraredCollateralVault.sol#L134C1-L159C6

Internal Pre-conditions

Vault asset with < 18 decimals

External Pre-conditions

N/A

Attack Path

After initial mint, all subsequent minting will result in significantly lower amount of shares for the same amount of deposit (1e12 times lower for 6 decimals asset). Taking the example from the Root Cause section - second depositor would be able to claim 100e6/(100e18 + 100e6) of 200e6 (actual deposited balance) which is 0.

Impact

This bricks the whole vault as all depositors except the first one would lose their funds. There are two possible first depositors - the protocol team minting shares to prevent vault inflation attack or normal user. In both cases the issue persists and the vault cant function normally.

PoC

N/A

Mitigation

Consider adjusting the asset amount to 18 decimals when doing shares/asset conversion.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/180

Issue M-8: CollVaultRouter::redeemToOne will lead to permanent loss of funds for users

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/30

Found by

vinica_boy

Summary

`CollVaultRouter::redeemToOne()` is designed to redeem shares of a collateral vault and swap all reward tokens to a single target one and send it to the redeemer. This works well if the collateral vault does not accrue iBGT rewards. In case it accrues iBGT rewards, those rewards are staked and the collateral vault accounts for the minted iBGTVault shares as reward and when user directly redeem through the collateral vault, those iBGTVault shares would be redeemed and rewards from the iBGTVault would be send directly to the redeemer.

In the case of redeeming through `CollVaultRouter` only rewards token for the collateral vault which we redeem against would be send to the user. If iBGTVault has different rewards they would become stuck in the `CollateralVaultRouter`.

Root Cause

Lets go through the workflow of harvesting rewards and the workflow of redeeming shares in a `InfraredCollateralVault` (excluding iBGTVault from the example as it has slightly different mechanism for rewards accrual).

In `InfraredCollateralVault::_harvestRewards()` we have an `_autoCompoundHook` to account for iBGT reward:

```
...
        (rewards, _token) = _autoCompoundHook(_token, _ibgt, _ibgtVault, rewards);
        uint fee = rewards * _performanceFee / BP;
        uint netRewards = rewards - fee;

        if (_token == asset()) {
          iVault.stake(netRewards);
          }
         // Meanwhile the token doesn't has an oracle mapped, it will be processed
↪  as a donation
        // This will avoid returns meanwhile a newly Infrared pushed reward token
↪  is not mapped
        if (_hasPriceFeed(_token) && _token != _iRedToken && !_isCollVault(_token))
↪  {
            _increaseBalance(_token, netRewards);
```

```
            // First time the oracle happens to be mapped, we add the token to the
↪   rewardedTokens
            // If token has no oracle map this won't be called, hence not DOS the
↪   vault at `totalAssets`
            _addRewardedToken(_token); // won't add duplicates
        }
...
```

and in `KodiakIslandVault` we see the implementation:

```
function _autoCompoundHook(address _token, address _ibgt, IIBGTVault _ibgtVault,
↪   uint _rewards) internal override returns (uint, address) {
    uint bbIbgtMinted;
    bool isIBGT = _token == _ibgt;
    if (isIBGT) {
        IERC20(_ibgt).safeIncreaseAllowance(address(_ibgtVault), _rewards);
        bbIbgtMinted = _ibgtVault.deposit(_rewards, address(this));
        _rewards = bbIbgtMinted;
    }
    _token = isIBGT ? address(_ibgtVault) : _token;

    return (_rewards, _token);
}
```

We see that when we have iBGT reward, it is staked into iBGTVault and the minted shares are accounted as a reward token.

When users redeem directly against the collateal vault, there are two internal mechanisms to withdraw the assets:

```
_withdraw(msg.sender, receiver, _owner, assetAmount, shares);
_withdrawExtraRewardedTokens(receiver, netShares, _totalSupply);
```

The first one handles the underlying asset of the vault and the second one handles all rewarded tokens.

Taking a look into `_withdrawExtraRewardedTokens()`, we see that the iBGTVault shares are not directly sent to the redeemer, but they are redeemed and user gets the rewards from iBGTVault.

```
if (token == _ibgtVault && _ibgtVault != address(this)) {
        IInfraredCollateralVault(token).redeem(amount, receiver, address(this));
} else {
     IERC20(token).safeTransfer(receiver, amount);
}
```

In the context of `CollVaultRouter`, the receiver of all tokens initially is the router:

```
params.collVault.redeem(params.shares, address(this), msg.sender);
```

This means that both reward tokens groups (from the collateral vault which we redeem against and the reward tokens from iBGTVault) will be send to the router but only tokens from the initial collateral vault would be swapped to target token and send to the user because we only construct the `tokens` array based on the initial collateral vault rewards:

```
address[] memory tokens = params.collVault.tryGetRewardedTokens();
```

`_harvestRewards()`: https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/InfraredCollateralVault.sol#L82C1-L129C6 `_autoCompoundHook()`: https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/KodiakIslandVault.sol#L27C1-L38C6 `redeemToOne`: https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/periphery/CollVaultRouter.sol#L352C1-L393C6

Internal Pre-conditions

Reward tokens of the collateral vault which we redeem against is a subset of the reward tokens for iBGTVault

External Pre-conditions

N/A

Attack Path

Consider the case where:

- `KodiakIslandVault` has reward tokens A, B and iBGTVault (actually `iBGT`, but it is staked an accounted as `iBGTVault`).
- `iBGTVault` has reward tokens A, B, C.
- User use `CollateralVaultRouter::redeemToOne()`
  - `KodiakIslandVault` transfers A, B as reward tokens of itself and A, B, C as reward tokens from `iBGTVault`
  - `tokens` array which is constructed at the beginning of the `redeemToOne()` workflow will only contain A, B, iBGTVault
  - amount of received iBGTVault would be zero because it was internally redeemed for A, B, C
- Only A and B tokens would be swapped to target token which would be send to user, leaving token C stuck.

There is an admin functionality `claimLockedTokens` which can rescue those stuck funds but there is way for attacker to use those funds before they are saved. In `depositFromAny()` we separately provide the `inputTokenAmount` and the `dexCalldata` and `dexCalldata` can be arbitrary constructed by users. In normal workflow, it is expected that there are no funds in the router contract and the specified input token amount for the OogaBooga swap will be provided by the user. Attacker can take advantage of the stuck funds and use them for his deposit by specifying an `inputAmount` for his `dexCalldata` equal to `params.inputAmount` + the amount of stuck funds. This way attacker does not need to provide all the funds used for the actual swap and will utilize the funds that previous redeemers have lost due to the mismatch of reward tokens.

Adding reference for the swap API of OogaBooga Router:
https://docs.oogabooga.io/deployments/aggregator/obrouter-reference

Impact

Permanent loss of funds for users using `redeemToOne` functionality.

PoC

N/A

Mitigation

When constructing the `tokens` array in `redeemToOne()`, consider using an intersection of the rewards set of both iBGTVault and the collateral vault shares which are redeemed. This is okay solution only if it is sure that the only collateral vault reward which is accepted is iBGTVault.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/177

Issue M-9: Users can redeem more `CollateralToken` by splitting a single `debtAmount` into multiple smaller `debtAmount`

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/34

The protocol has acknowledged this issue.

Found by

newspacexyz

Summary

Users can redeem more `CollateralToken` by splitting a single `debtAmount` into multiple smaller `debtAmount`..

Root Cause

The `_updateBaseRateFromRedemption` function increases the `baseRate` proportionally to `collateralDrawn`:

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/DenManager.sol#L498-L513

In DenManager.sol#L692, `collateralFee` is calculated by the increased `baseRate`.

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/DenManager.sol#L687-L692

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

No response

Impact

Users can redeem more `CollateralToken` by splitting a single `debtAmount` into multiple smaller `debtAmount`.

PoC

In first case for single, `redeemCollateral` function is called with `_debtAmount = 0.5e18`. In second case for multiple, `redeemCollateral` function is called 10 times with `_debtAmount = 0.05e18`.

```
function test_redeemCollaterals_test_single() external {
        _openDen(depositor); // opens and adds 1.58 wBERA
        _addCollateralToDen(1e18); // adds 1 wBERA

        vm.warp(block.timestamp + beraborrowCore.dmBootstrapPeriod()); // 1 day
↪   after the bootstrap period
        //deal(address(wBERA), depositor, 1e18);

        uint prevDepositorNectarBalance = nectarToken.balanceOf(depositor);
        uint prevDepositorBeraBalance = IERC20(wBERA).balanceOf(depositor);

        uint prevFeeReceiverBeraBalance =
↪   IERC20(wBERA).balanceOf(beraborrowCore.feeReceiver());

        uint debtForRedeem = 0.5e18;

        (address firstRedemptionHint, uint256 partialRedemptionHintNICR,) =
↪   multiCollateralHintHelpers.getRedemptionHints(denManager, debtForRedeem,
↪   priceFeed.fetchPrice(address(wBERA)), 0);

        vm.prank(depositor);

        denManager.redeemCollateral({
            _debtAmount: debtForRedeem,
            _firstRedemptionHint: firstRedemptionHint,
            _upperPartialRedemptionHint: address(0),
            _lowerPartialRedemptionHint: address(0),
            _partialRedemptionHintNICR: partialRedemptionHintNICR,
            _maxIterations: uint256(0), // 0 means no limit
            _maxFeePercentage: 6e17 // 60%
            // even that has been no redemptions in 2 weeks, since the redemption
↪   is 100% of the debt, the fee is 50% (huge), the baseRate is 50%, while
↪   redemptionFeeFloor is 0.5%
        });

        uint newDepositorNectarBalance = nectarToken.balanceOf(depositor);
        uint newDepositorBeraBalance = IERC20(wBERA).balanceOf(depositor);
```

```
        uint newFeeReceiverBeraBalance =
↪   IERC20(wBERA).balanceOf(beraborrowCore.feeReceiver());

        console2.log("--- single ---");

        console2.log("Depositor NectarBalance Decrease", prevDepositorNectarBalance
↪   - newDepositorNectarBalance);
        console2.log("Depositor BeraBalance Increase", newDepositorBeraBalance -
↪   prevDepositorBeraBalance);
        console2.log("FeeReceiver BeraBalance Increase", newFeeReceiverBeraBalance
↪   - prevFeeReceiverBeraBalance);

    }

    function test_redeemCollaterals_test_multi() external {
        _openDen(depositor); // opens and adds 1.58 wBERA
        _addCollateralToDen(1e18); // adds 1 wBERA

        vm.warp(block.timestamp + beraborrowCore.dmBootstrapPeriod()); // 1 day
↪   after the bootstrap period

        uint prevDepositorNectarBalance = nectarToken.balanceOf(depositor);
        uint prevDepositorBeraBalance = IERC20(wBERA).balanceOf(depositor);

        uint prevFeeReceiverBeraBalance =
↪   IERC20(wBERA).balanceOf(beraborrowCore.feeReceiver());

        uint debtForRedeem = 0.5e18;

        for(uint i = 0; i < 10; i ++) {

            (address firstRedemptionHint, uint256 partialRedemptionHintNICR,) =
↪   multiCollateralHintHelpers.getRedemptionHints(denManager, debtForRedeem / 10,
↪   priceFeed.fetchPrice(address(wBERA)), 0);

            vm.prank(depositor);

            denManager.redeemCollateral({
                _debtAmount: debtForRedeem / 10,
                _firstRedemptionHint: firstRedemptionHint,
                _upperPartialRedemptionHint: address(0),
                _lowerPartialRedemptionHint: address(0),
                _partialRedemptionHintNICR: partialRedemptionHintNICR,
                _maxIterations: uint256(0), // 0 means no limit
                _maxFeePercentage: 6e17 // 60%
                // even that has been no redemptions in 2 weeks, since the
↪   redemption is 100% of the debt, the fee is 50% (huge), the baseRate is 50%,
↪   while redemptionFeeFloor is 0.5%
            });
```

```
            vm.stopPrank();
        }

        uint newDepositorNectarBalance = nectarToken.balanceOf(depositor);
        uint newDepositorBeraBalance = IERC20(wBERA).balanceOf(depositor);
        uint newFeeReceiverBeraBalance =
↪   IERC20(wBERA).balanceOf(beraborrowCore.feeReceiver());

        console2.log("--- multi ---");

        console2.log("Depositor NectarBalance Decrease", prevDepositorNectarBalance
↪   - newDepositorNectarBalance);
        console2.log("Depositor BeraBalance Increase", newDepositorBeraBalance -
↪   prevDepositorBeraBalance);
        console2.log("FeeReceiver BeraBalance Increase", newFeeReceiverBeraBalance
↪   - prevFeeReceiverBeraBalance);

    }
```

--- single --- Depositor NectarBalance Decrease 500000000000000000 Depositor BeraBalance Increase 373121890547263682 FeeReceiver BeraBalance Increase 126878109452736318

--- multi --- Depositor NectarBalance Decrease 500000000000000000 Depositor BeraBalance Increase 415201653496957928 FeeReceiver BeraBalance Increase 84798346503042072

CollateralDrawn_single = 373121890547263682 CollateralDrawn_multi = 415201653496957928

415201653496957928 - 373121890547263682 = 42,079,762,949,694,246 42,079,762,949,694,246 * 100 / 373121890547263682 = 11.2%

Excess token amount is 11.2% of `CollateralDrawn_single`.

Mitigation

Check the logic of `_updateBaseRateFromRedemption` or Set a minimum `_debtAmount` to prevent manipulation of small redeem.

Issue M-10: `InternalizeDonations` will never work for `asset` token

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/43

Found by

bughuntoor

Summary

If any tokens are donated to the `InfraredCollateralVault`, owner can call `internalizeDonations` and create a linear vesting for the newly donated funds.

The exact donated amount is calculated in the following line:

```
uint donatedAmount = IERC20(token).balanceOf(address(this)) -
↪  getBalanceOfWithFutureEmissions(token);
require(donatedAmount >= amount, "CollVault: insufficient balance");
```

It subtracts the virtual balance from the actual token balance and treats it as donation. The problem when this asset is the main asset, is that usually the token balance would be 0, while the virtual accounting would be high. This is because all deposited assets get staked directly in the Infrared Vault. This would make the line above revert due to underflow and if any funds are sent as donation, they'd simply remain stuck.

Same issue exists also in `receiveDonations`

Root Cause

Logic issue

Impact

Loss of funds/ Stuck funds, Broken functionality

Affected Code

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/InfraredCollateralVault.sol#L237

Mitigation

Usually if there's any `asset` balance within the contract, it should be treated as donation

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/176

Issue M-11: `InfraredCollateralVault::rebalance()` will DoS the protocol if rebalancing more than the current available balance

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/56

Found by

0x73696d616f, 0xeix

Summary

`InfraredCollateralVault::rebalance()` decreases a currency balance for the `asset()`. However, the decrease in the sent currency is not capped to the current unlocked balance, allowing balance allocated to future emissions to be used.

An attacker can force this by withdrawing some funds in the currency sent used to rebalance, before the admin rebalances, in order to make the rebalance send too many sent currency and DoS the protocol.

Root Cause

In `InfraredCollateralVault:180`, rebalance(), there is no cap in the sent currency to rebalance.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Attacker frontruns admin rebalance with `withdraw()` call in the token that is going to be rebalanced by an admin.
2. The rebalance call withdraws too many token, leading to an underflow when calculating the unlocked balance in `totalAssets()`, 1, 2, 3 DoSing the protocol.

Impact

All protocol functions are DoSed due to the underflow above.

PoC

See above.

Mitigation

Check if the amount to rebalance is smaller than the current unlocked balance.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/174

Issue M-12: Non EIP712 compliance in `LiquidationManager::batchLiquidateDensWithPermit()`

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/71

The protocol has acknowledged this issue.

Found by

0x73696d616f, vinica_boy

Summary

`LiquidationManager::batchLiquidateDensWithPermit()` is not EIP712 compliant as it does not hash the `_denArray` to include in the `structHash`, as per the EIP:

> The dynamic values bytes and string are encoded as a keccak256 hash of their contents.

Root Cause

In `LiquidationManager:599`, `_denArray` is not hashed.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Call `LiquidationManager::batchLiquidateDensWithPermit()`.

Impact

Non EIP712 compliance.

PoC

Code

Mitigation

Hash the `_denArray`.

Issue M-13: Missing slippage control for certain router actions

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/91

Found by

0x73696d616f

Summary

Router actions such as `CollVaultRouter::redeemCollateralVault()` only check slippage when the `CollVault` asset is unwrapped. However, the amount of collateral to redeem depends on the price of the collateral, which means the amount of collateral redeemed may vary a lot and the user has no control over it.

Root Cause

Router actions such as `CollVaultRouter::redeemCollateralVault()` do not validate the amount of collateral redeemed if it is not unwrapped.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. User calls `CollVaultRouter::redeemCollateralVault()` without unwrapping and loses significant collateral due to a price drop in the meantime.

Impact

Loss of funds.

PoC

CollVaultRouter::redeemCollateralVault() DenManager::_removeCollateralFromDen()

Mitigation

Always place slippage control whenever there is a price.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/172

Issue M-14: `BoycoVault::totalAssets()` uses `asset.balanceOf()` instead of using virtual balances, allowing donation attacks

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/117

The protocol has acknowledged this issue.

Found by

0x73696d616f, TessKimy

Summary

`BoycoVault::totalAssets()` calls `super.totalAssets();`, which is the ERC4626 function, that simply does `return $._asset.balanceOf(address(this));`. As such, it's possible to do donation attacks. The comment indicates a variable should be used to track the balance, but this is not the case.

```
/// @dev Virtual accounting to avoid donations, asset valued denomination, returned in WAD
```

Root Cause

`BoycoVault:186` does not use virtual accounting.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

Simple donation attack due to not using virtual accounting, that is, attacker mints 1 share and before a user deposits assets, the attacker donates assets and inflates the share price, making the vulnerable user loss all or part of its deposit due to rounding down.

Impact

Loss of funds.

PoC

ERC4626Upgradeable

```
function totalAssets() public view virtual returns (uint256) {
    ERC4626Storage storage $ = _getERC4626Storage();
    return $._asset.balanceOf(address(this));
}
```

Mitigation

Use virtual accounting.

Issue M-15: Double Performance Fee on Donations When Oracle is Added

Source: https://github.com/sherlock-audit/2025-01-boyco-judging/issues/125

Found by

Afriaudit

Summary

When a token without an oracle is donated, it's treated as a donation without fees. Later, when an oracle is mapped to the token, internalizing the donation incorrectly applies the performance fee a second time, reducing the escrowed funds twice.

Root Cause

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/InfraredCollateralVault.sol#L109

https://github.com/sherlock-audit/2025-01-boyco/blob/main/blockend/src/core/vaults/InfraredCollateralVault.sol#L244

Duplicate Fee Logic: Fees are applied during both donation and internalization.

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

without oracle harvested without an oracle mapped. Map Oracle: Oracle is assigned Internalize Donation: Protocol processes the donation again. Apply Fee Twice: Performance fees are deducted both during donation and internalization.

Impact

double fee applied reducing harvest

PoC

No response

Mitigation

dont apply fee for token with no oracle mapped

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Beraborrowofficial/blockend/pull/175

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.