SHERLOCK

Security Review For Beraborrow

Collaborative Audit Prepared For:	**Beraborrow**
Lead Security Expert(s):	**pkqs90** **santipu_**
Date Audited:	**May 19 - May 24, 2025**
Final Commit:	**6981534**

Introduction

This security review is a deep audit into BEX Price Feeds, Leverage, Collateral Vaults and LSP Routers.

Scope

Repository: Beraborrowofficial/blockend

Audited Commit: fdc2180abad7bd65bffdf8d32b817ba9df8235dc

Final Commit: 6981534ce5098a52a31402883fa4d08ed9d05ee7

Files:

- src/core/spotOracles/BPTStableFeed.sol
- src/core/spotOracles/BPTWeightedFeed.sol
- src/periphery/CollVaultRouter.sol
- src/periphery/DeleverageRouter.sol
- src/periphery/LSPRouter.sol
- src/periphery/LeverageRouter.sol

Final Commit Hash

6981534ce5098a52a31402883fa4d08ed9d05ee7

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.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	11	11

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: DoS When Deploying BPT Price Feeds Due to Empty Storage

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/7

Summary

The contracts `BPTStableOracle` and `BPTWeightedFeed` cannot be deployed because a constructor check always reverts.

Vulnerability Detail

In the constructors of these contracts, the following check will always fail:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/core/spotOracles/BPTStableFeed.sol#L52-L54

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/core/spotOracles/BPTWeightedFeed.sol#L59-L61

```
if (PRICE_FEED.fetchPrice(tokens[0].token) == 0 ||
↪  PRICE_FEED.fetchPrice(tokens[1].token) == 0) {
    revert NoFeed();
}
```

At the point of this check, the tokens struct in storage has not yet been initialized, so `tokens[0].token` and `tokens[1].token` both default to `address(0)`. Attempting to fetch a price for `address(0)` will revert because it is not a valid token.

Impact

Deployment of `BPTStableOracle` and `BPTWeightedFeed` is blocked, resulting in a denial of service.

Recommendation

Move the price check to occur only after the tokens struct has been properly initialized with valid token addresses.

Discussion

santipu03

Fixed, now the price check is performed with the `_tokens` array, which is passed as an argument in the constructor.

Issue M-2: DoS When Deploying BPTStableFeed Due to Incorrect Token Ordering

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/8

Summary

The `BPTStableFeed` contract cannot be deployed if the token order from the Balancer Vault differs from the contract's assumptions.

Vulnerability Detail

During deployment, the following check is performed:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/core/spotOracles/BPTStableFeed.sol#L48-L50

```
if (_tokens[0] != address(poolTokens[0]) || _tokens[1] != address(poolTokens[2])) {
    revert InvalidPoolTokens();
}
```

This check assumes that `getPoolTokens` from the Balancer Vault will return the two tokens at fixed positions (indices 0 and 2). However, the Balancer Vault does not guarantee this ordering – token addresses are sorted when stored.

If the expected tokens are not in these exact positions, the check will revert, preventing the deployment of `BPTStableFeed` for affected Balancer Stable Pools.

Impact

Deployment of `BPTStableFeed` fails, causing a denial of service.

Recommendation

Remove the strict position-based check and instead validate that the input tokens match the two tokens present in the Balancer Stable Pool, regardless of their ordering.

Discussion

santipu03

Fixed, now the check only ensures that the tokens are part of the `poolTokens` array.

Issue M-3: Stuck Funds When Closing Den With DeleverageRouter

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/10

Summary

When closing a Den using `DeleverageRouter`, users unintentionally lose the debt gas compensation, which gets stuck in the router contract until governance manually claims it.

Vulnerability Detail

In Beraborrow, when a user closes a Den, they are only required to repay the debt minus the gas compensation. However, the `DeleverageRouter` contract ignores this and forces users to repay the full debt, including the gas compensation.

This results in excess funds (equal to the gas compensation) being left behind in the router contract, effectively trapping them until governance intervenes.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocken d/src/periphery/DeleverageRouter.sol#L89

```
  function closeDen(IDenManager denManager, DenLoopingParams calldata
↪   closeDenWithFlashLoanParams)
      external
      nonReentrant
  {
      (, uint256 debt) = denManager.getDenCollAndDebt(msg.sender);

      // ...

>>    uint debtToLoan = debt - closeDenWithFlashLoanParams.nectProvidedByUser;
  }
```

Impact

Users closing Dens via `DeleverageRouter` will permanently lose the amount set aside as debt gas compensation, as it remains locked in the contract.

Recommendation

Update the logic to subtract the debt gas compensation from `debtToLoan` so users only repay what is strictly required.

```
        (, uint256 debt) = denManager.getDenCollAndDebt(msg.sender);

        if (debt == 0) revert NoDenToClose();

-       if (closeDenWithFlashLoanParams.nectProvidedByUser >= debt) revert
 ↪  NoNeedToLoop();
+       uint256 debtGasCompensation = borrowerOperations.DEBT_GAS_COMPENSATION();

+       if (closeDenWithFlashLoanParams.nectProvidedByUser >= debt -
 ↪  debtGasCompensation) revert NoNeedToLoop();

        if (closeDenWithFlashLoanParams.nectProvidedByUser != 0 ) {
            nect.transferFrom(msg.sender, address(this),
 ↪  closeDenWithFlashLoanParams.nectProvidedByUser);
        }

-       uint debtToLoan = debt - closeDenWithFlashLoanParams.nectProvidedByUser;
+       uint debtToLoan = debt - closeDenWithFlashLoanParams.nectProvidedByUser -
 ↪  debtGasCompensation;
```

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue M-4: Reverting Functions on LSPRouter Due to 0-Balance Tokens in CollVaults

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/14

Summary

Many functions in `LSPRouter` can revert if a `CollVault` contains a reward token with little or no balance, due to mismatches between previewed and actual token outputs.

Vulnerability Detail

Most LSPRouter functions follow this flow:

1. **Preview Phase**: Simulate a withdrawal using `collVaultRouter.previewRedeemUnderlying` to get the array of underlying reward tokens.
2. **Execution Phase**: Perform the actual withdrawal using `_withdrawUnderlyingCollVaultAssets`.
3. **Validation Phase**: Compare the previewed token array with the actual withdrawn tokens, expecting them to match exactly.

Example: `redeemToOne`:

1. First, it calls `previewRedeem`, which calls `_previewWithdrawUnderlyingCollVaultAssets`, which in turn ends up calling `collVaultRouter.previewRedeemUnderlying` to simulate the vault withdrawal and get the array of underlying reward tokens.

```
    function redeemToOne(
        ILSPRouter.RedeemToOneParams calldata params
    ) external returns (uint assets, uint totalAmountOut) {
        address[] memory tokensToClaim = lspUnderlyingTokens();
        Arr memory arr = _initArr(tokensToClaim, address(this));
>>      (, address[] memory tokens,) = previewRedeem(params.shares);
```

3. When executing the actual withdrawal, it calls `_withdrawUnderlyingCollVaultAssets`, which gets the actual reward tokens the user will receive.

```
_withdrawUnderlyingCollVaultAssets(
    collVault,
    amount,
    _tokens
);
```

5. Finally, it checks that the two arrays of underlying reward tokens match exactly.

```
      uint[] memory swapAmounts = new uint[](tokens.length);
      for (uint i; i < tokens.length; i++) {
>>        if (tokens[i] != _tokens[i]) revert TokenPreviewMismatch(i, _tokens[i],
 ↪  tokens[i]);
          swapAmounts[i] = IERC20(_tokens[i]).balanceOf(address(this)) -
 ↪  underlyingCurrAmounts[i];
      }
```

The issue here is that this final check will often fail due to the discrepancies in how the reward tokens are added in the preview and in the actual execution.

During the preview, the function `collVaultRouter.previewRedeemUnderlying` will skip any reward tokens that have little to no balance, because it means the user won't be able to swap those:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/CollVaultRouter.sol#L436C1-L440C44

```
uint256 tokenBalance = vault.getBalance(token) + v.earned;
if (tokenBalance == 0) continue;

uint256 tokenAmount = v.netShares.mulDiv(tokenBalance, v.totalSupply,
 ↪  Math.Rounding.Down);
if (tokenAmount == 0) continue;
```

But during the actual execution of the withdrawal, the function `_withdrawUnderlyingCollVaultAssets` won't skip any tokens, even if they have no balance in their `CollVaults`.

This discrepancy will cause both arrays to be different, causing a DoS on most of the functions on `LSPRouter`.

Impact

The functions impacted are the following:

- `redeemToOne`
- `redeemPreferredUnderlyingToOne`
- `_processUnderlyingTokens`

Given that `_processUnderlyingTokens` is an internal function, it will affect all the functions calling it, which are the following:

- `redeemPreferredUnderlying`
- `redeem`
- `withdraw`

Recommendation

Update `_withdrawUnderlyingCollVaultAssets` to skip tokens that would result in zero balance for the user, ensuring alignment between preview and execution flows.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue M-5: DoS on _processUnderlyingTokens due to empty _tokens array

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/15

Summary

The `_processUnderlyingTokens` function in `LSPRouter` will always revert when `unwrap` is set to false because of a faulty comparison between an empty array and a populated one. This creates a denial-of-service (DoS) condition for key withdrawal and redemption functions.

Vulnerability Detail

The `_processUnderlyingTokens` function is called by `redeemPreferredUnderlying`, `redeem`, and `withdraw` to process the underlying tokens a user should receive after withdrawing from the LSP.

When a user opts not to unwrap their tokens on withdrawal, the function is expected to transfer the underlying assets directly from the LSP to the user, bypassing the `CollVault` unwrapping logic.

However, a check in the code wrongly compares the `tokens` array (passed in as input, fully populated with expected token addresses) with the local `_tokens` array, which remains empty if `unwrap` is false:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/LSPRouter.sol#L317

```
  function _processUnderlyingTokens(
      Arr memory arr,
      address[] memory tokens,
      address[] memory tokensToClaim,
      uint[] memory underlyingCurrAmounts,
      address receiver,
      bool unwrap
  ) internal returns (uint[] memory amounts) {
>>    address[] memory _tokens = new address[](tokens.length);
      uint[] memory _amounts = new uint[](tokens.length);
      uint[] memory currAmounts = tokensToClaim.underlyingAmounts(arr.receiver);
      bool firstCollVaultFound;

      if (unwrap) {
          // ...
      } else {
          amounts = new uint[](tokens.length);
          for (uint i; i < tokens.length; i++) {
```

```
>>             if (tokens[i] != _tokens[i]) revert TokenPreviewMismatch(i,
 ↪  _tokens[i], tokens[i]);
               amounts[i] = currAmounts[i] - arr.prevAmounts[i];
           }
       }
   }
```

The issue arises because the `_tokens` array is only filled inside the `if (unwrap)` block. When `unwrap` is false, `_tokens` stays uninitialized (effectively all-zero addresses), causing the comparison `tokens[i] != _tokens[i]` to always fail and revert.

This makes the entire function permanently unusable in `unwrap = false` cases, creating a denial-of-service (DoS) vulnerability across all dependent functions.

Impact

This bug causes a permanent DoS when users attempt to withdraw without unwrapping. As a result, the following functions are also effectively broken:

- `redeemPreferredUnderlying`
- `redeem`
- `withdraw`

These functions will always revert under non-unwrap conditions, blocking key user interactions.

Recommendation

Remove the redundant `if (tokens[i] != _tokens[i])` check, as it serves no meaningful purpose when `unwrap` is false and guarantees a revert.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue M-6: _simulateWithdraw Miscalculates Redeemed Amounts from LSP

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/16

Summary

The `_simulateWithdraw` function is intended to simulate the amounts a user will receive after withdrawing from the LSP. However, its internal calculation logic is flawed, leading to incorrect values being returned. This discrepancy can cause transaction reverts or user fund losses when the previewed values are later used for actual swaps.

Vulnerability Detail

The functions `previewRedeemToOne` and `previewRedeem` both rely on `_simulateWithdraw` to estimate the token amounts a user would receive upon withdrawal.

However, `_simulateWithdraw` does not replicate the actual logic used by the LSP when calculating withdrawals. Specifically, the function currently computes:

```
sharesToRedeem * assetVirtualBalance / totalVirtualBalance
```

as seen here:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/LSPRouter.sol#L535C1-L552C6

```
function _simulateWithdraw(uint assets, address[] memory tokens)
    internal
    view
    returns (uint[] memory expectedAmounts)
{
    expectedAmounts = new uint[](tokens.length);
    uint totalVirtual;

    for (uint i; i < tokens.length; i++) {
        totalVirtual += lspGetters.getTokenVirtualBalance(tokens[i]);
    }
    require(totalVirtual > 0, "No virtual balance");

    for (uint i; i < tokens.length; i++) {
        uint tokenVirtualBalance = lspGetters.getTokenVirtualBalance(tokens[i]);
        expectedAmounts[i] = assets * tokenVirtualBalance / totalVirtual;
    }
}
```

In contrast, the actual LSP withdrawal logic calculates:

```
(sharesToRedeem - fee) * assetVirtualBalance / totalSupply
```

This mismatch between the preview and actual behavior leads to significant inconsistencies in the returned amounts array.

Impact

Because `previewRedeemToOne` and `previewRedeem` use these incorrect simulated values to construct swap transactions in `redeemToOne`, two main risks arise:

1. **Overestimation**: The amounts array contains values higher than the actual amounts, causing the swap to attempt using tokens that the contract never received, resulting in a revert.
2. **Underestimation**: The amounts array contains values lower than the actual amounts, leading the swap to handle fewer tokens than available, causing a loss of funds for the user.

Recommendation

Update the `_simulateWithdraw` function to align its calculations with the actual LSP withdrawal logic, including correctly accounting for fees and using `totalSupply` rather than `totalVirtualBalance`. This will ensure that the previewed amounts accurately reflect the real outcomes and avoid downstream errors.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue M-7: CollVaultRouter.sol::_simulateVault Redemption doesn't correctly handle iBGTVault as reward token.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/20

Summary

For a InfraredCollateralVault that has iBGTVault as it's reward token, the `CollVaultRouter.sol::_simulateVaultRedemption` does not correctly calculate the amount of iBGT rewards.

Vulnerability Detail

When `vault != ibgtVault` and `token == v.iBGT`, it means we are handling the iBGTVault shares (because iBGTVault is converted to iBGT in `tryGetRewardedTokensIncludingIbgtVault()` function).

The tokenBalance should reflect the amount of iBGTVault shares, which would be passed on to next level recursion. However, the current implementation incorrectly sets tokenBalance to the amount of iBGT in vault.

The total amount of iBGTVault shares should be `vault.getBalance(iBGTVault) + ibgtVault.previewDeposit(v.earned)`.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/CollVaultRouter.sol#L420-L432

```
try vault.infraredVault() returns (IInfraredVault infraredVault) {
    if (token == v.ibgt) {
        // Internal earned amount not included, returned amount will probably be
↪   slightly lower
        v.earned = infraredVault.earned(address(vault), v.ibgt);
        v.earned -= v.earned * v.performanceFee / BP;
        // No `autoCompoundHook` on iBGTVault
        if (vault == ibgtVault) {
            v.earned = ibgtVault.previewDeposit(v.earned);
        }
    } else {
        v.earned = infraredVault.earned(address(vault), token);
        v.earned -= v.earned * v.performanceFee / BP;
    }
} catch {
    v.earned = 0;
}
uint256 tokenBalance = vault.getBalance(token) + v.earned;
```

```
if (tokenBalance == 0) continue;

uint256 tokenAmount = v.netShares.mulDiv(tokenBalance, v.totalSupply,
↪  Math.Rounding.Down);
if (tokenAmount == 0) continue;
```

Impact

Inaccurate iBGT vault shares for previewRedeem. This will especially be an issue if previewRedeem is used to build swap params.

Recommendation

```
if (token == v.ibgt) {
    tokenBalance = vault.getBalance(address(ibgtVault)) +
↪  ibgtVault.previewDeposit(v.earned);
}
```

Issue M-8: Token array mismatch in CollVaultRouter.sol::redeemToOne() and CollVaultRouter.sol::previewRedeemUnderlying.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/21

Summary

`CollVaultRouter.sol::previewRedeemUnderlying` returns an array of token and amounts that user should receive when redeeming.

`CollVaultRouter.sol::redeemToOne()` requires user input a swap param for each token received during redeem.

If the `token[]` arrays mismatch, the swap would fail.

Vulnerability Detail

The `redeemToOne` function fetches the `tokens[]` array by functions `tryGetRewardedTokens()` and `tryGetRewardedTokensIncludingIbgtVault()`. These two functions are view-only functions, and does not check for zero-balance for user.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/CollVaultRouter.sol#L308

```
    function redeemToOne(
        RedeemToOneParams calldata params
    ) external {
@>      address[] memory tokens = params.collVault.tryGetRewardedTokens();
@>      (address[] memory rewardTokens, uint length) =
↪   tokens.tryGetRewardedTokensIncludingIbgtVault(params.collVault.asset(),
↪   ibgtVault);
        ...

        for (uint i; i < length; i++) {
            address token = rewardTokens[i];

            uint amount = IERC20(token).balanceOf(address(this)) - prevBalances[i];
            if (token != params.targetToken) {
                if (amount > 0 && params.tokensSwapCalldatas[i].length != 0) {
                    IERC20(token).safeIncreaseAllowance(params.swapRouter, amount);

                    SwappersLib.executeSwap(swapperData, params.swapRouter,
↪   params.tokensSwapCalldatas[i]);
                }
            } else {
                IERC20(token).safeTransfer(params.receiver, amount);
```

```
            }
        }
```

The `_simulateVaultRedemption` function, which is used for previewing redeem, skips adding the token if there is zero reward for the token.

This inconsistency behavior would cause the return value for `previewRedeemUnderlying()` be unusable for assembling swap param inputs when calling `redeemToOne()`.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/CollVaultRouter.sol#L437

```
    function _simulateVaultRedemption(
        IInfraredCollateralVault vault,
        uint sharesToRedeem,
        DynamicArrayLib.DynamicArray memory tokens,
        DynamicArrayLib.DynamicArray memory amounts,
        bool isNested
    ) internal view {
        ...
        for (uint256 i; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];

            ...
            uint256 tokenBalance = vault.getBalance(token) + v.earned;
@>          if (tokenBalance == 0) continue;

            uint256 tokenAmount = v.netShares.mulDiv(tokenBalance, v.totalSupply,
↪   Math.Rounding.Down);
@>          if (tokenAmount == 0) continue;

            // Recursively simulate if token is a nested vault
            // We check 'token == v.ibgt' instead of ibgtVault since its replaced
↪   by ibgt in tryGetRewardedTokensIncludingIbgtVault
            // And iBGT can only be on rewarded tokens of a vault that it's not
↪   iBGTVault in the form of iBGTVault
            if (token == v.ibgt && token != address(ibgtVault)) {
                _simulateVaultRedemption(ibgtVault, tokenAmount, tokens, amounts,
↪   true);
            } else {
                TokenValidationLib.aggregateIfNotExistent(token, tokenAmount,
↪   tokens, amounts);
            }
        }
    }
```

Impact

If the `token[]` arrays mismatch for `redeemToOne()`, the swap would fail.

Recommendation

Two recommendations for `_simulateVaultRedemption()` (the second would be more robust).

1. Keep the code as it is, and remove the skip zero balance logic.
2. Construct the token balance array in the beginning (using the same method as `redeemToOne()`: by `tryGetRewardedTokens()` and `tryGetRewardedTokensIncludingIbgtVault()`), and fill in amounts. This way we sure the token order is correct.

Another recommendation to improve robustness of `redeemToOne()`: Have users pass a `tokens[]` array as input for validation, to make sure swap params are in the correct order. Or even better, parse the token directly from swap params.

Issue M-9: LeverageRouter.sol::calculateDebtAmount() cannot set target leverage and desiredCR at the same time.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/22

Summary

`LeverageRouter.sol::calculateDebtAmount()` cannot set target `leverage` and `desiredCR` at the same time.

Vulnerability Detail

The purpose for `LeverageRouter.sol::calculateDebtAmount()` is unclear. The definition of `leverage` here is the value of collateral, compared to the margin. The definition of `desiredCR` here is collateral / debt. Because we have:

- `collateral = margin + debt = leverage * margin`
- `CR = collateral / debt = leverage * margin / (leverage * margin - leverage) = leverage / (leverage - 1)`

It is not possible to specify the final state of both `leverage` and `desiredCR`, we can only pick one. For example, if `margin = 100, desiredCR = 1.2`, then we will always have `collateral = 600, debt = 500` and `leverage = 6`. If `margin = 100, leverage = 3`, then we will always have `collateral = 300, debt = 200` and `leverage = 1.5`.

If this function is aiming to set the target `leverage` and only using `desiredCR` as a constraint from preventing leverage to be too high, the formula on for `compositeDebt` should be changed to `uint256 compositeDebt = marginInCollVault * collVaultPrice * (leverage - BP) / BP;`

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknend/src/periphery/LeverageRouter.sol#L138

```
    function calculateDebtAmount(
        IDenManager denManager,
        address position,
        uint256 margin,
        uint256 leverage,
        uint256 desiredCR,
        bool isRecoveryMode
    ) external view returns (uint256 debtAmount) {
        _checkValidDesiredCR(denManager, desiredCR, isRecoveryMode);

        uint256 maxLeverage = calculateMaxLeverage(desiredCR);
```

```
        if (leverage > maxLeverage) {
            revert LeverageExceeded(leverage, maxLeverage);
        }

        // Get the collVault of DenManager
        address collVault = denManager.collateralToken();
        // Get the current collateral amount and debt of position in DenManager
        (uint256 currentColl,) = denManager.getDenCollAndDebt(position);

        // Get the collVault shares of margin (margin is the amount of collVault's
↪   asset)
        uint256 marginInCollVault =
↪   IInfraredCollateralVault(collVault).previewDeposit(margin);

        uint256 collVaultPrice = priceFeed.fetchPrice(collVault);

        // compositeDebt includes BorrowingFee and GasCompensation
@>      uint256 compositeDebt = marginInCollVault * collVaultPrice * leverage /
↪   (desiredCR * BP);

        uint256 borrowingRate = isRecoveryMode ? 0 :
↪   denManager.getBorrowingRateWithDecay();

        uint256 debtGasCompensation = currentColl == 0 ?
↪   borrowerOperations.DEBT_GAS_COMPENSATION() : 0;
        compositeDebt -= debtGasCompensation;

        if (compositeDebt < borrowerOperations.minNetDebt()) {
            revert DebtTooLow(compositeDebt, borrowerOperations.minNetDebt());
        }

        debtAmount = compositeDebt * WAD / (WAD + borrowingRate);
    }
```

Impact

The function will return an incorrect debtAmount, which would be used by users to operate Den.

Recommendation

Only set `leverage` as target, and change `desiredCR` to `minimumCR` to a constraint.

Issue M-10: In LeverageRouter.sol, user margin should be used directly as collateral rather then repaying flashloan.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/23

Summary

In `LeverageRouter.sol`, user margin should be used directly as collateral rather then repaying flashloan.

Vulnerability Detail

The current design for user margin is: user supplies margin as `collVault.asset()`, and is swapped to NECT for repaying the flashloan.

Another design is to directly deposit the margin into Den (need to deposit `collVault.asset() -> collVault` first), and purely use the borrowed NECT for repaying flashloan. This way, we can avoid swap slippage from `collVault.asset() -> NECT`.

Also, `calculateDebtAmount()` function assumes we put the margin into Den collateral (it assumes `CR = (debt + margin) / debt`), which aligns with the above implementation.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknend/src/periphery/LeverageRouter.sol#L183

```
function _processFlashLoan(
    Action action,
    address denManager,
    DenLoopingParams memory params,
    address account,
    uint256 amount
) private {
    address collVault = IDenManager(denManager).collateralToken();
    address asset = IInfraredCollateralVault(collVault).asset();

    uint256 collVaultSharesMinted = _swapAndWrap(params.nectToColl, asset,
↪   collVault, amount);

    if (action == Action.OpenDen) {
        _openDen(denManager, account, collVault, params.denParams,
↪   collVaultSharesMinted);
    } else {
        _increaseCollateral(denManager, account, collVault, params.denParams,
↪   collVaultSharesMinted);
    }
```

```
    _handleRepayment(
        account, asset, amount, params.marginCollAmount, params.collToNect,
↪   params.denParams.debtAmount
    );
}
```

Impact

1. Unnecessary swap slippage from `collVault.asset() -> NECT`.
2. Implementation does not align with `calculateDebtAmount()` view function.

Recommendation

Provided above.

Issue M-11: BPT uses fair-pricing, opening up arbitrage opportunities for redemption.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/27

Summary

BPT uses fair-pricing, opening up arbitrage opportunities for redemption.

Vulnerability Detail

In Beraborrow, BPT can be used as collateral to borrow Nect. The BPT pricing we use is fair-value pricing, to prevent price manipulation, which is good. However, this opens up arbitrage opportunities using redemption if the pool is imbalanced.

Let's assume the pool is (100 USDC, 100 DAI). Half of the total BPT supply would equal to (50 USDC, 50 DAI), which is 100 USD.

Now, assume naturally the pool goes imbalanced to 80 USDC, 125 DAI. Usually, arbitragers on chain will come and rebalance the pool, but let's assume arbitragers hadn't done that yet. Now, half of the total BPT supply, in fair pricing, is still 100 USD (because fair-pricing does not change on pool imbalance). However, if we exit the liquidity, the BPT is worth 40 USDC + 62.5 DAI = 102.5 USD.

This means if the pool is imbalanced, anyone can redeem the BPT at fair-pricing, and exit liquidity to earn arbitrage. The more imbalanced the pool is, the more profitable it is.

Note that this is not an *exploit*, because for normal users who hold BPT, this scenario is equivalent to if they were just holding on to their BPT and they didn't exit liquidity when it was imbalanced, regardless of whether they put it in Beraborrow or not. All they *lose* is the opportunity cost.

What beraborrow does is it provides a way for arbitragers to take advantage of the users by redemption, and can take away the arbitraging opportunity the user had for themselves.

Impact

If the pool is imbalanced, anyone can redeem the BPT at fair-pricing, and exit liquidity to earn arbitrage.

The most extreme scenario is, if the pool was extremely imbalanced, then arbitragers can redeem all BPT tokens for themselves.

Recommendation

To prevent arbitraging, disable the redemption for BPT tokens.

Issue L-1: Warnings of BPTStableFeed usage

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/9

Summary

The `BPTStableFeed` contract is designed to always take the minimum value between the two tokens in the BEX pool, which could open arbitrage or attack opportunities during a depeg event.

Vulnerability Detail

To calculate the fair price of a stable BPT, `BPTStableFeed` uses the lower bound between the two token values:

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/core/spotOracles/BPTStableFeed.sol#L65C1-L92C6

```
  function _calculatePrice() internal view returns (uint256) {
      TokenInfo[2] storage tokenInfo = tokens;

      uint256 min = type(uint256).max;

      // 18 decimals
      uint256[2] memory prices = _getTokenPrices([tokenInfo[0].token,
↪  tokenInfo[1].token]);

      try BEX_POOL.getTokenRate(address(tokenInfo[0].token)) returns (uint256 rate)
↪  {
          prices[0] = prices[0].divWad(rate);
      } catch {}

      if (prices[0] < min) {
>>        min = prices[0];
      }

      try BEX_POOL.getTokenRate(address(tokenInfo[1].token)) returns (uint256 rate)
↪  {
          prices[1] = prices[1].divWad(rate);
      } catch {}

      if (prices[1] < min) {
>>        min = prices[1];
      }

      uint256 poolRate = BEX_POOL.getRate();
```

```
        return min.mulWad(poolRate);
    }
```

This design ensures users cannot borrow against the maximum price (which could lead to over-borrowing) and instead values BPT conservatively.

However, during a depeg event, when one of the pool's tokens loses its peg, the oracle will undervalue BPT relative to its spot price. This can open unfair liquidation or redemption opportunities, creating profitable arbitrage paths at the expense of the protocol and its users.

To reduce risk, we recommend the following:

1. **Guardian/Bot Monitoring** Deploy a guardian or bot to continuously monitor BPT prices. If a depeg or abnormal pricing is detected, the system should have the ability to pause redemptions on affected DenManager contracts.
2. **User Warnings** Clearly inform users using stable BPT as collateral that the oracle may misprice during depeg events. If this oracle design is used in production, users must be aware of the risks.
3. **Understand BEX Dependencies** Keep in mind that BPT value depends on BEX's security. Any risky protocol changes or exploits in BEX (especially if diverging from Balancer) can directly affect NECT's stability.
4. **Strict Borrow Limits** Set conservative borrow caps in DenManagers that accept stable BPT as collateral. If something goes wrong, the protocol's maximum exposure should be capped to limit damage and protect overall NECT stability.

Discussion

alex-beraborrow

This can only open unfair liquidations, but not redemption opportunities since it would price the collateral to the downside. @santipu03

santipu03

> This can only open unfair liquidations, but not redemption opportunities since it would price the collateral to the downside.

If collateral is priced lower than its real value, redemptions will give out too much collateral, allowing redeemers to make some profit from Beraborrow users.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/core/DenManager.sol
allowbreak #L714

```
    function _redeemCollateralFromDen(
        ISortedDens _sortedDensCached,
        address _borrower,
```

```
        uint256 _maxDebtAmount,
        uint256 _price,
        address _upperPartialRedemptionHint,
        address _lowerPartialRedemptionHint,
        uint256 _partialRedemptionHintNICR
    ) internal returns (SingleRedemptionValues memory singleRedemption) {
        Den storage t = Dens[_borrower];
        // Determine the remaining amount (lot) to be redeemed, capped by the
↪   entire debt of the Den minus the liquidation reserve
        singleRedemption.debtLot = BeraborrowMath._min(_maxDebtAmount, t.debt -
↪   DEBT_GAS_COMPENSATION);

        // Get the CollateralLot of equivalent value in USD
>>      singleRedemption.collateralLot = (singleRedemption.debtLot *
↪   DECIMAL_PRECISION) / _price;

        // ...
    }
```

Issue L-2: Function _checkValidDesiredCR Is Missing Edge Cases

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/11

Summary

The `_checkValidDesiredCR` function is designed to validate whether a user's desired collateral ratio (CR) is acceptable, but it currently misses important edge cases, leading to potential downstream reverts.

Vulnerability Detail

When users call `calculateDebtAmount`, the desired CR is validated using `_checkValidDesiredCR`. This function ensures the final individual collateral ratio (ICR) is above the minimum collateral ratio (MCR) or the critical collateral ratio (CCR), depending on whether the system is in Recovery Mode.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/LeverageRouter.sol#L302C1-L312C6

```
function _checkValidDesiredCR(IDenManager denManager, uint256 desiredCR, bool
↪  isRecoveryMode) private view {
    if (isRecoveryMode) {
        if (desiredCR < beraborrowCore.CCR()) {
            revert CollateralRatioBelowCCR();
        }
    } else {
        if (desiredCR < denManager.MCR()) {
            revert CollateralRatioBelowMCR();
        }
    }
}
```

However, this check misses two edge cases:

1. **Normal Mode (non-Recovery)**: The system-wide total collateral ratio (TCR) must not drop below CCR when the new debt is added.
2. **Recovery Mode**: The user's new ICR must not be lower than their current ICR; otherwise, the leverage operation will revert later.

Without accounting for these, users may pass the initial CR check but encounter unexpected reverts when attempting to leverage their Den.

Impact

Users are misled into thinking their desired CR is valid, only to encounter reverts during the leverage operation, causing wasted gas and a poor user experience.

Recommendation

Enhance `_checkValidDesiredCR` to explicitly handle:

1. The TCR check in non-Recovery Mode, ensuring the system remains above CCR.
2. The ICR improvement check in Recovery Mode, ensuring the user's new ICR exceeds their current one.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue L-3: Function calculateMaxLeverage Does Not Consider Existing Positions

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/12

Summary

The `calculateMaxLeverage` function is designed to compute the maximum allowable leverage based on a user's desired collateral ratio (CR). However, it only accounts for new Den openings and fails to handle cases where the user is adjusting an existing Den, leading to incorrect results.

Vulnerability Detail

When determining the amount of debt a user needs to flash loan for leveraging, the system calls `calculateDebtAmount`, which internally relies on `calculateMaxLeverage` to check if the desired leverage is valid.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocken d/src/periphery/LeverageRouter.sol#L179C1-L181C6

```
function calculateMaxLeverage(uint256 desiredCR) public pure returns (uint256
↪  maxLeverage) {
    maxLeverage = desiredCR * BP / (desiredCR - WAD);
}
```

This formula assumes the user is opening a new Den and ignores any existing collateral or debt. As a result, when a user tries to further leverage an existing Den, the function returns an incorrect maximum leverage value. This leads to reverts later in the leverage process, even though the system initially signals that the parameters are valid.

Impact

Users may be misled into thinking their leverage adjustments are valid, only to encounter reverts when attempting to increase leverage on an existing Den. This wastes gas and harms the user experience.

Recommendation

Update `calculateMaxLeverage` to incorporate the user's current collateral and debt when calculating the maximum leverage, ensuring it accurately reflects both new and existing Den scenarios.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue L-4: claimCollateralRouter Can Be Used to Sweep Funds From CollVaultRouter

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/13

Summary

An attacker can exploit the `claimCollateralRouter` function by passing in a malicious `denManager` contract, allowing them to sweep collateral tokens left in the `CollVaultRouter` contract.

Vulnerability Detail

The `CollVaultRouter` contract provides the `claimCollateralRouter` function, allowing users to claim collateral from any `DenManager` and redeem it through the `CollVault`.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/periphery/CollVaultRouter.sol#L252C1-L264C6

```
function claimCollateralRouter(
    IDenManager denManager,
    IInfraredCollateralVault collVault,
    address receiver,
    uint minAssetsWithdrawn
) external {
    uint surplusBalance = denManager.surplusBalances(msg.sender);

    denManager.claimCollateral(msg.sender, address(this));

    uint assetsWithdrawn = collVault.redeem(surplusBalance, receiver,
↪   address(this));
    require(assetsWithdrawn >= minAssetsWithdrawn, "assetsWithdrawn <
↪   _minAssetsWithdrawn");
}
```

The issue arises because the `denManager` input is not validated. An attacker can supply a malicious `denManager` contract that:

1. Returns an arbitrary `surplusBalance` value equal to (or targeting) the leftover collateral tokens in the router.
2. Does nothing in its `claimCollateral` call, avoiding any balance changes.
3. Allows the router to proceed to `collVault.redeem()`, pulling out the targeted collateral and sending the underlying assets to the attacker.

This effectively lets anyone sweep all remaining collateral from the `CollVaultRouter`.

Impact

An attacker can drain any leftover collateral tokens sitting in the `CollVaultRouter` contract, resulting in a direct loss of funds.

Recommendation

Add strict validation to ensure that the provided `denManager` is an authorized and valid `DenManager` registered within the Beraborrow system.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue L-5: Potential DoS When Swapping Zero Amounts

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/17

Summary

The `_swapToTargetToken` function in `LSPRouter` attempts to perform swaps even when the token amount is zero. Depending on the implementation of the external Swapper contract, this can cause reverts, leading to a denial-of-service (DoS) condition in critical redemption functions.

Vulnerability Detail

The `_swapToTargetToken` function is invoked by both `redeemToOne` and `redeemPreferredUnderlyingToOne` to convert the various received reward tokens into a single target token for the user.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/LSPRouter.sol#L354C1-L380C6

```
    function _swapToTargetToken(
        address targetToken,
        ILSPRouter.SwapAllTokensToOneParams memory params,
        address[] memory _tokens,
        uint[] memory _amounts
    ) private returns (uint targetTokenAmountOut) {
        for (uint i; i < tokensLength; i++) {
            uint amount = _amounts[i];
            address _token = _tokens[i];
            if (_token != targetToken) {
                if (params.tokensSwapCalldatas[i].length > 0) {

↪   IERC20(_token).safeIncreaseAllowance(address(params.swapRouter), amount);

>>                  SwappersLib.executeSwap(swapperData, params.swapRouter,
↪   params.tokensSwapCalldatas[i]);
                }
            } else {
                IERC20(targetToken).safeTransfer(params.receiver, amount);
            }
        }
    }
```

The issue is that even when amount is zero, the function still proceeds to:

1. Approve the swap router for zero tokens, and
2. Call SwappersLib.executeSwap.

Depending on how the Swapper contract handles zero-amount swaps, this can trigger a revert. For example, if the swap logic contains a require(amount > 0) or similar input check, the entire transaction will fail.

This creates a vulnerability where a zero balance on any reward token can block the whole redemption process.

Impact

A zero balance in any reward token can:

1. Cause a revert during the swap step.
2. Result in a full DoS on redeemToOne and redeemPreferredUnderlyingToOne.

This means users will be unable to complete their withdrawals if the router holds zero for any expected reward token.

Recommendation

Add a conditional check to skip the swap step when `amount == 0`.

Discussion

santipu03

Fixed by applying the recommended mitigation.

Issue L-6: Simplify code in CollVaultRouter.sol::_simulateVaultRedemption

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/18

Summary

The `CollVaultRouter.sol::_simulateVaultRedemption` handles a lot of edge cases, which can be simplified. Not a security issue, just to improve code maintainability.

Vulnerability Detail

1. https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/periphery/CollVaultRouter.sol#L445C1-L449C14

Since v.iBGT is never equal to ibgtVault, the following check can be changed from

```
@>          if (token == v.ibgt && token != address(ibgtVault)) {
                _simulateVaultRedemption(ibgtVault, tokenAmount, tokens, amounts,
↪   true);
            } else {
                TokenValidationLib.aggregateIfNotExistent(token, tokenAmount,
↪   tokens, amounts);
            }
```

to

```
if (token == v.ibgt) {
    _simulateVaultRedemption(ibgtVault, tokenAmount, tokens, amounts, true);
} else {
    TokenValidationLib.aggregateIfNotExistent(token, tokenAmount, tokens, amounts);
}
```

2. https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/periphery/CollVaultRouter.sol#L420-L432

Since when the current `vault` is `ibgtVault`, we will always have `v.asset == v.ibgt`. This means the `if (vault == ibgtVault)` check in the try-catch clause is redundant, because `token == v.ibgt` would've been already handled in the above `if (token == v.asset)` check and skipped.

Also, the `v.earned = ibgtVault.previewDeposit(v.earned);` does not make much sense, since when `vault = iBGTVault`, we want to calculate the earning rewards in iBGT and not vault shares.

It would be cleaner to change the code from

```
            // Skip if it's the primary asset
            if (token == v.asset) {
                TokenValidationLib.aggregateIfNotExistent(v.asset, v.assetAmount,
↪   tokens, amounts);
                continue;
            }

            try vault.infraredVault() returns (IInfraredVault infraredVault) {
                if (token == v.ibgt) {
                    // Internal earned amount not included, returned amount will
↪   probably be slightly lower
                    v.earned = infraredVault.earned(address(vault), v.ibgt);
                    v.earned -= v.earned * v.performanceFee / BP;
                    // No `autoCompoundHook` on iBGTVault
@>                  if (vault == ibgtVault) {
                        v.earned = ibgtVault.previewDeposit(v.earned);
                    }
                } else {
                    v.earned = infraredVault.earned(address(vault), token);
                    v.earned -= v.earned * v.performanceFee / BP;
                }
            } catch {
                v.earned = 0;
            }
```

to

```
// Skip if it's the primary asset
// This handled the case where vault = iBGTVault and asset = iBGT.
if (token == v.asset) {
    TokenValidationLib.aggregateIfNotExistent(v.asset, v.assetAmount, tokens,
↪   amounts);
    continue;
}

try vault.infraredVault() returns (IInfraredVault infraredVault) {
    v.earned = infraredVault.earned(address(vault), token);
    v.earned -= v.earned * v.performanceFee / BP;
} catch {
    v.earned = 0;
}
```

Impact

Improve code maintainability.

Recommendation

Provided above.

Issue L-7: User can pass arbitrary params.denManager to CollVaultRouter.sol.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/19

Summary

In CollVaultRouter.sol, user needs to provide the address for `params.denManager`. However, for some functions, `params.denManager` is not validated, and can be arbitrary addresses.

Vulnerability Detail

1. Take function `redeemCollateralVault()` as an example. We only check that `params.collVault` is valid, but does not check `params.denManager`.

The same issue exists for `closeDenVault()` and `claimCollateralRouter()`.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/periphery/CollVaultRouter.sol#L214

```
    function redeemCollateralVault(
        ICollVaultRouter.RedeemCollateralVaultParams memory params
    ) external {
@>      require(_isWhitelistedCollateralAt(address(params.collVault),
↪   params.collIndex), "Incorrect collateral");
@>      require(address(params.denManager.collateralToken()) ==
↪   address(params.collVault), "Incorrect DenManager or Vault");

        uint prevSharesBalance = params.collVault.balanceOf(address(this));
        uint prevNectBalance = nectar.balanceOf(address(this));

        nectar.sendToPeriphery(msg.sender, params._debtAmount);

        params.denManager.redeemCollateral(
            params._debtAmount,
            params._firstRedemptionHint,
            params._upperPartialRedemptionHint,
            params._lowerPartialRedemptionHint,
            params._partialRedemptionHintNICR,
            params._maxIterations,
            params._maxFeePercentage
        );
        ...
    }
```

2. The `_validateVaultAndManager()` function is used to validate user inputs for many functions, e.g. `openDenVault()`, `adjustDenVault()`. Howver, it also does not validate

denManager.

Suggest to add `require(borrowerOperations.denManagers(index) == denManager);`.

```
function _validateVaultAndManager(
    IInfraredCollateralVault collVault,
    uint _collIndex,
    IDenManager denManager
) internal view returns(IERC20 vaultAsset) {
    require(_isWhitelistedCollateralAt(address(collVault), _collIndex), "Incorrect
↪   collateral");
    require(address(denManager.collateralToken()) == address(collVault), "Incorrect
↪   DenManager or Vault");
    vaultAsset = IERC20(collVault.asset());
}
```

Impact

Did not find a way to perform direct attacks. But better add the validation, just to be safe.

Recommendation

1. Call `_validateVaultAndManager()` for all entry functions, e.g. `redeemCollateralVault()`, `closeDenVault()` and `claimCollateralRouter()`.
2. Validate `denManager` for `_validateVaultAndManager()`, as suggested above.

Issue L-8: Redundant code in LSPRouter.sol.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/24

Summary

In `LSPRouter.sol` there are some redundant code, e.g. duplicate functions, unused functions. It can be simplified for better code maintainability.

Vulnerability Detail

1. redeemToOne() vs. redeemPreferredUnderlyingToOne().

Two functions are 95% alike, the only difference is

1. Fetching the `tokens[]` from different preview functions. One uses `previewRedeem()`, the other uses `previewRedeemPreferredUnderlying()`.
2. Performing the actual redeem. One uses `lsp.redeem(params.shares, arr.receiver, msg.sender);`, the other uses `lsp.redeem(params.shares, params.preferredUnderlyingTokens, arr.receiver, msg.sender);`.

Suggest to:

1. Combine the two functions, or move the shared code to an independent function.
2. Add clear comments on the difference between these two functions. The current comment for `redeemPreferredUnderlyingToOne()` is inaccurate.

```
    // @audit: This comment is wrong.
@>  /// @notice First token of `params.preferredUnderlyingTokens` is the target
↪   token, if it's a collVault, it's the asset token of the collVault
    /// @dev Auto unwraps and swaps all to the target token
    function redeemPreferredUnderlyingToOne(
        ILSPRouter.RedeemPreferredUnderlyingToOneParams calldata params
    ) external returns (uint assets, uint totalAmountOut) {
        ...
    }
```

2. previewRedeemToOne() vs. previewRedeem().

The two functions are exactly the same, and `previewRedeemToOne()` is unused anywhere.

Suggest to remove `previewRedeemToOne()` function.

Impact

Improve code maintainability.

Recommendation

Provided above.

Issue L-9: Redundant tokens[] array check in redeem(), withdraw(), redeemPreferredUnderlying().

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/25

Summary

Redundant `tokens[]` array check in `redeem()`, `withdraw()`, `redeemPreferredUnderlying()`.

Vulnerability Detail

The three functions `redeem()`, `withdraw()`, `redeemPreferredUnderlying()` does not need to perform a swap, they only redeem/withdraw the vault shares and unwrap underlying tokens to rewards.

Currently, in these functions, they also perform a `tokens[]` array check, expecting the preview function's `tokens[]` array order to be exactly the same as when we are unwrapping the CollVault shares to reward tokens.

This check is redundant, and even if the array order does not match (since it also depends on another router `CollVaultRouter::previewRedeem()`), it shouldn't block users from redeeming.

The `tokens[]` array order check for swap params may make sense, since we expect the array params to be in the same order. But for these non-swap functions, it is not required.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blockend/src/periphery/LSPRouter.sol#L228

```
function redeem(
    ILSPRouter.RedeemWithoutPrederredUnderlyingParams calldata params
) external returns (uint assets, address[] memory tokens, uint[] memory amounts) {
    address[] memory tokensToClaim = lspUnderlyingTokens();
    Arr memory arr = _initArr(tokensToClaim, params.unwrap ? address(this) :
↪   params.receiver);
    uint[] memory underlyingCurrAmounts;
    if (params.unwrap) {
        /// @dev List of tokens we expect to receive after unwrapping all collVaults
        (, tokens,) = previewRedeem(params.shares);
        /// @dev Called bore lsp::redeem
        underlyingCurrAmounts = tokens.underlyingAmounts(arr.receiver);
    } else {
        tokens = tokensToClaim;
    }

    // Call redeem without preferred tokens
    assets = lsp.redeem(params.shares, arr.receiver, msg.sender);
```

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

    amounts = _processUnderlyingTokens(arr, tokens, tokensToClaim,
↪  underlyingCurrAmounts, params.receiver, params.unwrap);
}

function _processUnderlyingTokens(
    Arr memory arr,
    address[] memory tokens,
    address[] memory tokensToClaim,
    uint[] memory underlyingCurrAmounts,
    address receiver,
    bool unwrap
) internal returns (uint[] memory amounts) {
    address[] memory _tokens = new address[](tokens.length);
    uint[] memory _amounts = new uint[](tokens.length);
    uint[] memory currAmounts = tokensToClaim.underlyingAmounts(arr.receiver);
    bool firstCollVaultFound;

    if (unwrap) {
        for (uint i; i < tokensToClaim.length; i++) {
            uint amount = currAmounts[i] - arr.prevAmounts[i];
            if (amount > 0) {
                if (priceFeed.isCollVault(tokensToClaim[i])) {
                    if (!firstCollVaultFound) {
                        _addIbgtVaultRewardTokens(_tokens);
                        firstCollVaultFound = true;
                    }
                    IInfraredCollateralVault collVault =
↪  IInfraredCollateralVault(tokensToClaim[i]);

                    _withdrawUnderlyingCollVaultAssets(
                        collVault,
                        amount,
                        _tokens
                    );
                } else {
                    _aggregateIfNotExistentWithoutAmounts(_tokens,
↪  tokensToClaim[i]);
                }
            }
        }
        for (uint i; i < tokens.length; i++) {
            if (tokens[i] != _tokens[i]) revert TokenPreviewMismatch(i, _tokens[i],
↪  tokens[i]);
            _amounts[i] = IERC20(_tokens[i]).balanceOf(address(this)) -
↪  underlyingCurrAmounts[i];
            IERC20(_tokens[i]).safeTransfer(receiver, _amounts[i]);
        }
```

```
        amounts = _amounts;
    } else {
        amounts = new uint[](tokens.length);
        for (uint i; i < tokens.length; i++) {
            if (tokens[i] != _tokens[i]) revert TokenPreviewMismatch(i, _tokens[i],
↪   tokens[i]);
            amounts[i] = currAmounts[i] - arr.prevAmounts[i];
        }
    }
}
```

Impact

Simplify the code by a lot.

Recommendation

1. Don't perform preview (e.g. `previewRedeem()`, `previewRedeemPreferredUnderlying()`) in these functions.
2. Drop the `tokens[]` array order check in `_processUnderlyingTokens()`.

Issue L-10: Design recommendation: tokens[] order check for redeemToOne() and redeemPreferred UnderlyingToOne() should be changed.

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/26

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Design recommendation: `tokens[]` order check for `redeemToOne()` and `redeemPreferredUnderlyingToOne()` should be changed.

Vulnerability Detail

`redeemToOne()` and `redeemPreferredUnderlyingToOne()` functions perform swaps after redeeming the underlying tokens. Since the underlying tokens are subject to change (e.g. depending on vesting, new rewards added, etc.), we need to make sure the swap params provided to the function matches the token we are swapping.

The current code first queries the `previewRedeem` function and fetches a `tokens[]` array. Then it does the actual redeem and aggregates the received tokens to `_tokens[]` array. At last, it checks the `tokens[]` and `_tokens[]` should match exactly.

The flaw of this design is:

1. Even if the two token arrays match, we are still unsure whether the swap params matches the token, because the swap params is passed in by `params.tokensSwapCalldatas`. It depends on how the offchain swap param is assembled.
2. The `tokens[]` and `_tokens[]` are calculated using different code, while handling a lot of edge cases (e.g. skipping zero balance), it is hard to get them 100% correct.

My suggestion here would be to change the `tokens[]` array from `preview received tokens vs. actual received token` to `token in swap params vs. actual received token`. If we can decode the input token from `tokensSwapCalldatas`, it would be a more robust solution.

Since we are using the `EnsoRebalancer.sol` as the swapper, we can directly decode the input token from the first parameter.

Note that even if we don't want to decode the token from swap params (e.g. due to supporting different swappers), we can always enforce users to pass in a `tokens[]` array that they expect to match the swap params, and use that to validate against received tokens.

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocknd/src/periphery/swappers/EnsoRebalancer.sol#L60

```
    function swap(
@>      address _tokenIn,
        uint256 _amountIn,
        address _tokenOut,
        bytes calldata _data
    ) external {
        // Decode swap payload
        (SwapPayload memory payload, uint minTargetTokenReceived) =
↪   abi.decode(_data, (SwapPayload, uint));

        if (_tokenIn == ibgtVault) {
            _redeemToOne(_tokenIn, _amountIn, payload, _tokenOut,
↪   minTargetTokenReceived);
        } else {
            _singleSwap(_tokenIn, _amountIn, payload, _tokenOut);
        }

    }
```

https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/blob/main/blocken d/src/periphery/LSPRouter.sol#L82

```
    struct RedeemToOneParams {
        uint shares;
        address receiver;
        address swapRouter;
        IInfraredCollateralVault collVault;
        address targetToken;
        uint minTargetTokenAmount;
        bytes[] tokensSwapCalldatas;
    }

    function redeemToOne(
        ILSPRouter.RedeemToOneParams calldata params
    ) external returns (uint assets, uint totalAmountOut) {
        address[] memory tokensToClaim = lspUnderlyingTokens();
        Arr memory arr = _initArr(tokensToClaim, address(this));
@>      (, address[] memory tokens,) = previewRedeem(params.shares);
        uint[] memory underlyingCurrAmounts =
↪   tokens.underlyingAmounts(arr.receiver);

        assets = lsp.redeem(params.shares, arr.receiver, msg.sender);
        require(assets >= params.minAssetsWithdrawn, "LSPRouter: assetsWithdrawn <
↪   minAssetsWithdrawn");
        address[] memory _tokens = new address[](tokens.length);
        bool firstCollVaultFound;

        uint[] memory currAmounts = tokensToClaim.underlyingAmounts(address(this));
```

```
        for (uint i; i < tokensToClaim.length; i++) {
            uint amount = currAmounts[i] - arr.prevAmounts[i];
            if (amount > 0) {
                if (priceFeed.isCollVault(tokensToClaim[i])) {
                    if (!firstCollVaultFound) {
                        _addIbgtVaultRewardTokens(_tokens);
                        firstCollVaultFound = true;
                    }
                    IInfraredCollateralVault collVault =
↪   IInfraredCollateralVault(tokensToClaim[i]);

                    _withdrawUnderlyingCollVaultAssets(
                        collVault,
                        amount,
                        _tokens
                    );
                } else {
                    _aggregateIfNotExistentWithoutAmounts(_tokens,
↪   tokensToClaim[i]);
                }
            }
        }

        uint[] memory swapAmounts = new uint[](tokens.length);
        for (uint i; i < tokens.length; i++) {
@>          if (tokens[i] != _tokens[i]) revert TokenPreviewMismatch(i, _tokens[i],
↪   tokens[i]);
            swapAmounts[i] = IERC20(_tokens[i]).balanceOf(address(this)) -
↪   underlyingCurrAmounts[i];
        }

        totalAmountOut = _swapToTargetToken(
            params.targetToken,
            ILSPRouter.SwapAllTokensToOneParams({
                receiver: params.receiver,
                minTargetTokenAmount: params.minTargetTokenAmount,
                swapRouter: params.swapRouter,
                tokensSwapCalldatas: params.tokensSwapCalldatas
            }),
            _tokens,
            swapAmounts
        );
    }
```

Impact

A more robust solution of validating swap parameter token order.

Recommendation

Decode the input token from swap params, and check it matches the `tokens[]` order for the actual redeem.

Discussion

alex-beraborrow

Enso calldata can't be decoded, it has different underlying structures depending on the DEX they're routing from

Issue L-11: Inconsistent Address Representation for Native Coin (BERA)

Source: https://github.com/sherlock-audit/2025-05-beraborrow-may-19th/issues/28

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `CollVaultRouter` and `LSPRouter` contracts use different placeholder addresses to represent the native coin (BERA), creating inconsistencies across the protocol.

Vulnerability Detail

Both `CollVaultRouter` and `LSPRouter` include a `claimLockedTokens` function that allows the protocol admin to recover tokens locked in the router contracts.

However, the two contracts use different conventions for representing the native coin:

1. `LSPRouter` uses `address(0)` to represent BERA.
2. `CollVaultRouter` uses `address(0xdead)` to represent BERA.

Recommendation

For clarity and consistency, standardize the placeholder address used to represent BERA across all contracts in the system.

Discussion

santipu03

Acknowledged.

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.