



Beraborrow Blockend Competition

July 8, 2025

Contents

1	Introduction	2
1.1	About Cantina	2
1.2	Disclaimer	2
1.3	Risk assessment	2
1.3.1	Severity Classification	2
2	Security Review Summary	3
3	Findings	4
3.1	Medium Risk	4
3.1.1	Wrong Check in <code>InfraredCollateralVault::rebalance</code> Prevents Owner from Swapping Assets When Locked Emissions Exist	4
3.1.2	Missing Slippage Parameters in <code>PermissionlessPSM</code>	5
3.1.3	Collateral gas compensation is not taken into account when computing TCR on <code>batchLiquidateDens</code>	5
3.1.4	<code>InfraredVault.getRewardForUser()</code> will break harvest rewards	7
3.1.5	The order of tokens returned by <code>CollVaultRouter.previewRedeemUnderlying()</code> is not the same as that used in <code>redeemToOne()</code>	9
3.1.6	<code>CollVaultRouter::previewRedeemUnderlying</code> is miscalculating all iBGT unclaimed rewards	11
3.1.7	All unclaimed rewards are not being factored in <code>previewRedeemUnderlying</code> due to wrong argument ordering	12
3.1.8	NECT is not distributed to validators on <code>ValidatorPool</code>	14
3.1.9	Malicious users can steal all USDC on the <code>PermissionlessPSM.sol</code> for free by utilizing the rounding issue in <code>previewMint(...)</code>	15
3.1.10	Ordered liquidations are always broken due to <code>BrimeDen</code>	17
3.1.11	Incorrect <code>nectMinted</code> calculation in <code>pPermissionlessPSM</code>	19
3.1.12	Sunsetted collateral will lower TCR and trigger recovery mode	20
3.1.13	Rounding can cause sum of individual den debts to be greater than <code>totalActiveDebt</code> causing underflow on withdrawals	21
3.1.14	Liquidation of protocol den will cause locked funds for <code>boyco den</code>	23

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

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.

From Feb 8th to Mar 7th Cantina hosted a competition based on [beraborrow-blockend](#). The participants identified a total of **60** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 14
- Low Risk: 29
- Gas Optimizations: 0
- Informational: 17

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

3 Findings

3.1 Medium Risk

3.1.1 Wrong Check in `InfraredCollateralVault::rebalance` Prevents Owner from Swapping Assets When Locked Emissions Exist

Submitted by [santipu](#), also found by [cccz](#) and [hash](#)

Severity: Medium Risk

Context: `InfraredCollateralVault.sol`#L192-L193

Description: The `rebalance` function in `InfraredCollateralVault` allows the owner to swap tokens held by the Vault for its main asset. This function includes a check intended to prevent swapping tokens that are still locked, since doing so could cause a revert in subsequent calls to `totalAssets`—leading to a temporary DoS on the Vault:

```
/// @dev Token out or received currency will always be the asset of the vault
function rebalance(IInfraredCollateralVault.RebalanceParams calldata p) external virtual onlyOwner {
    // ...

    uint received = IAsset(asset()).balanceOf(address(this)) - receivedCurrencyBalance;
    uint sent = sentCurrencyBalance - IAsset(p.sentCurrency).balanceOf(address(this));

    // if we were to rebalance locked emissions, a possible revert on subsequent `$.balanceOf` calls would
    // occur
    if (sent > getBalance(p.sentCurrency) - getLockedEmissions(p.sentCurrency)) revert
    // WithdrawalLockedEmissions(); // <<<

    // ...
}
```

However, this check is incorrect because it subtracts locked emissions a second time. The `getBalance` function already accounts for locked emissions, making the additional subtraction unnecessary:

```
/// @dev Doesn't include vesting amount, same as `getTokenVirtualBalance` in LSPGetters
function getBalance(address token) public view override returns (uint) {
    return _getBalanceData().balanceOf(token);
}
```

```
/**
 * @notice Returns the unlocked token emissions
 */
function balanceOf(BalanceData storage $, address token) internal view returns (uint256) {
    EmissionSchedule memory schedule = $.emissionSchedule[token];
    return $.balance[token] - lockedEmissions(schedule, unlockTimestamp(schedule));
}
```

As a result, when there are locked emissions, the owner cannot swap the intended amount of tokens because they are being counted twice.

Example scenario:

1. A Vault holds 100 wETH, with 50 unlocked and 50 still locked.
2. The owner wants to swap 25 wETH to the Vault's main asset.
3. Due to the incorrect check, the owner cannot swap any wETH because the locked portion (50 wETH) is subtracted twice.
 - `sent` = 25 wETH.
 - `getBalance(p.sentCurrency)` = 50 wETH.
 - `getLockedEmissions(p.sentCurrency)` = 50 wETH.
 - The condition fails because $25 > 50 - 50$.

Because the `rebalance` function allows the Vault to swap tokens for the main asset (and subsequently stake them for additional rewards), this bug reduces the Vault's potential yield. If the owner can't use `rebalance`, users may opt for other Vaults or protocols offering higher returns.

Recommendation: Fix the rebalance function so that it does not subtract the locked emissions twice:

```
/// @dev Token out or received currency will always be the asset of the vault
function rebalance(IInfraredCollateralVault.RebalanceParams calldata p) external virtual onlyOwner {
    // ...

    // if we were to rebalance locked emissions, a possible revert on subsequent `$.balanceOf` calls would
    ⇨ occur
-   if (sent > getBalance(p.sentCurrency) - getLockedEmissions(p.sentCurrency)) revert
⇨   WithdrawingLockedEmissions();
+   if (sent > getBalance(p.sentCurrency)) revert WithdrawingLockedEmissions();

    // ...
}
```

Beraborrow: Fix implemented in line 57 at contract [0xcF2787690CD7d00F2aB90022339492bf21e1DBD3](#).

3.1.2 Missing Slippage Parameters in PermissionlessPSM

Submitted by [santipu](#), also found by [hash](#)

Severity: Medium Risk

Context: [PermissionlessPSM.sol#L71](#), [PermissionlessPSM.sol#L89](#), [PermissionlessPSM.sol#L107](#), [PermissionlessPSM.sol#L125](#)

Description: The PermissionlessPSM contract allows users to exchange stablecoins for NECT at a fixed 1:1 rate, plus a protocol fee. This fee is determined by an external contract called `feeHook`. Below is an example of how the fee is calculated using `feeHook`:

```
function previewDeposit(address stable, uint stableAmount) public view returns (uint mintedNect, uint nectFee) {
    uint64 wadOffset = stables[stable];

    if (wadOffset == 0) revert NotListedToken(stable);

    uint grossMintedNect = stableAmount * wadOffset;

    uint fee = feeHook.calcFee(msg.sender, stable, grossMintedNect, IFeeHook.Action.DEPOSIT); // <<<
    fee = fee == 0 ? DEFAULT_FEE : fee;

    nectFee = grossMintedNect.feeOnRaw(fee);
    mintedNect = grossMintedNect - nectFee;
}
```

Currently, we do not have the implementation details of `feeHook`, so it is unclear whether the fee is calculated at a fixed rate or adjusted dynamically. If it is dynamic, users of `PermissionlessPSM` have no way to cancel a transaction when the resulting fee is unexpectedly high.

Example scenario:

1. Bob wants to exchange 100 USDC for 100 NECT, expecting a maximum fee of 0.5%.
2. After calling deposit, he only receives 95 NECT, implying the fee was actually 5%.
3. Bob would have preferred to revert the transaction and use a DEX instead.

In this situation, Bob overpays on fees due to the lack of any slippage mechanism, resulting in a potential loss of funds. Also, even though block times on Berachain are fast, its mempool is public. A MEV bot could potentially sandwich Bob's transaction and further increase the fee he effectively pays.

Recommendation: Because `PermissionlessPSM` does not strictly follow EIP-4626, it is recommended to add a slippage parameter that reverts the transaction if the resulting fee exceeds the user's expectation.

Beraborrow: Fix implemented (see `deposit/mint/withdraw/redeem` functions) in contract [0xB2F796FA30A8512C1D27a1853a9a1a8056b5CC25](#).

3.1.3 Collateral gas compensation is not taken into account when computing TCR on `batchLiquidateDens`

Submitted by [santipu](#), also found by [hash](#)

Severity: Medium Risk

Context: LiquidationManager.sol#L498, LiquidationManager.sol#L536-L538

Description: When performing a batch liquidation (`batchLiquidateDens`) in Recovery Mode, the TCR is re-computed after each individual liquidation to ensure that is still below CCR so that liquidations can keep happening. However, the variable `entireSystemColl` is not correctly tracked as it wrongly includes the collateral gas compensation amount from already liquidated Dens. This issue was discovered in the private Sherlock contest and has been correctly fixed on `liquidateDens`, but the function `batchLiquidateDens` still has this issue. Below we can see that `liquidateDens` has the issue fixed:

```
function liquidateDens(IDenManager denManager, uint256 maxDensToLiquidate, uint256 maxICR, address liquidator)
↳ public {
    // ...

    if (densRemaining > 0 && !denManagerValues.sunsetting && denCount > 1) {
        (uint entireSystemColl, uint entireSystemDebt) = borrowerOperations.getGlobalSystemBalances();
        entireSystemColl -= (totals.totalCollGasCompensation + totals.totalCollToSendToSP) *
        ↳ denManagerValues.price; // <<<
        entireSystemDebt -= totals.totalDebtToOffset;
        address nextAccount = denManagerValues.sortedDens.getLast();
        while (densRemaining > 0 && denCount > 1) {
            uint ICR = denManager.getCurrentICR(nextAccount, denManagerValues.price);
            if (ICR > maxICR) break;
            unchecked {
                --densRemaining;
            }
            address account = nextAccount;
            nextAccount = denManagerValues.sortedDens.getPrev(account);

            {
                uint256 TCR = BeraborrowMath._computeCR(entireSystemColl, entireSystemDebt);
                if (TCR >= borrowerOperations.BERABORROW_CORE().CCR() || ICR >= TCR) break;
            }

            singleLiquidation = _tryLiquidateWithCap(
                denManager,
                account,
                debtInStabPool,
                _getApplicableMCR(account, denManagerValues),
                denManagerValues.price
            );
            if (singleLiquidation.debtToOffset == 0) continue;
            debtInStabPool -= singleLiquidation.debtToOffset;
            entireSystemColl -= // <<<
                (singleLiquidation.collToSendToSP + singleLiquidation.collSurplus // <<<
                + singleLiquidation.collGasCompensation) * denManagerValues.price; // <<<
            entireSystemDebt -= singleLiquidation.debtToOffset;
            _applyLiquidationValuesToTotals(totals, singleLiquidation);
            unchecked {
                --denCount;
            }
        }
    }
    // ...
}
```

On the other hand, `batchLiquidateDens` still has the same issue:

```
function batchLiquidateDens(IDenManager denManager, address[] memory _denArray, address liquidator) public {
    // ..

    if (denIter < _denArray.length && denCount > 1) {
        // second iteration round, if we receive a den with ICR > MCR and need to track TCR
        (uint256 entireSystemColl, uint256 entireSystemDebt) = borrowerOperations.getGlobalSystemBalances();
        entireSystemColl -= totals.totalCollToSendToSP * denManagerValues.price; // <<<
        entireSystemDebt -= totals.totalDebtToOffset;
        while (denIter < _denArray.length && denCount > 1) {
            address account = _denArray[denIter];
            uint ICR = denManager.getCurrentICR(account, denManagerValues.price);
            unchecked {
                ++denIter;
            }

            uint applicableMCR = _getApplicableMCR(account, denManagerValues);
        }
    }
}
```

```

    if (ICR <= _LSP_CR_LIMIT) {
        singleLiquidation = _liquidateWithoutSP(denManager, account);
    } else if (ICR < applicableMCR) {
        singleLiquidation = _liquidateNormalMode(
            denManager,
            account,
            debtInStabPool,
            denManagerValues.sunsetting
        );
    } else {
        if (denManagerValues.sunsetting) continue;
        {
            uint256 TCR = BeraborrowMath._computeCR(entireSystemColl, entireSystemDebt);
            if (TCR >= borrowerOperations.BERABORROW_CORE().CCR() || ICR >= TCR) continue;
        }
        singleLiquidation = _tryLiquidateWithCap(
            denManager,
            account,
            debtInStabPool,
            applicableMCR,
            denManagerValues.price
        );

        if (singleLiquidation.debtToOffset == 0) continue;
    }

    debtInStabPool -= singleLiquidation.debtToOffset;
    entireSystemColl -= // <<<
        (singleLiquidation.collToSendToSP + singleLiquidation.collSurplus) * // <<<
        denManagerValues.price;
    entireSystemDebt -= singleLiquidation.debtToOffset;
    _applyLiquidationValuesToTotals(totals, singleLiquidation);
    unchecked {
        --denCount;
    }
}
}
// ...
}

```

Due to this issue, when liquidations occur via `batchLiquidateDens`, the computed TCR will be higher than expected, causing some Dens to avoid liquidation when they should be liquidated. Also, given that TCR will be inflated, some Dens will be unfairly liquidated given that its ICR should be above real TCR but will be below the inflated TCR.

Recommendation: To mitigate this issue is recommended to apply the same fix from `liquidateDens` to `batchLiquidateDens`.

Beraborrow: Fix implemented in line 498 at contract `0x965dA3f96dCBfcCF3C1d0603e76356775b5afD2E`.

3.1.4 `InfraredVault.getRewardForUser()` will break harvest rewards

Submitted by [cccz](#), also found by [santipu](#), [cccz](#) and [hash](#)

Severity: Medium Risk

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

Description: `InfraredCollateralVault._harvestRewards()` calls `InfraredVault.getReward()` to claim rewards in `InfraredVault`, and charges `performanceFee` and updates rewards based on the change in token balance before and after the call.

```

function _harvestRewards() internal override {
    if (block.timestamp == _getInfraredCollVaultStorage().lastUpdate) return;

    IInfraredVault iVault = infraredVault();
    (address[] memory tokens, uint tokensLength) = _getVaultRewardTokens();
    uint[] memory prevBalances = new uint[](tokensLength);

    for (uint i; i < tokensLength; i++) {
        prevBalances[i] = IERC20(tokens[i]).balanceOf(address(this));
    }

    // harvest rewards
    iVault.getReward();

    uint _performanceFee = getPerformanceFee();
    address _iRedToken = iRedToken();
    address _ibgt = ibgt();
    IIBGTVault _ibgtVault = IIBGTVault(ibgtVault());
    // re-stake iBGT, take performance fee and update accounting
    for (uint i; i < tokensLength; i++) {
        address _token = tokens[i];
        uint newBalance = IERC20(_token).balanceOf(address(this));
    }
}

```

The problem here is that anyone can call `InfraredVault.getRewardForUser()` to claim rewards into `InfraredCollateralVault` on behalf of `InfraredCollateralVault._harvestRewards()`. This causes the token balance change 0 in `InfraredCollateralVault._harvestRewards()`, thus not updating any rewards.

```

function getReward() public {
    getRewardForUser(msg.sender);
}
// ...
function getRewardForUser(address _user)
    public
    nonReentrant
    updateReward(_user)
{
    onReward();
    uint256 len = rewardTokens.length;
    for (uint256 i; i < len; i++) {
        address _rewardsToken = rewardTokens[i];
        uint256 reward = rewards[_user][_rewardsToken];
        if (reward > 0) {
            (bool success, bytes memory data) = _rewardsToken.call{
                gas: 200000
            }(
                abi.encodeWithSelector(
                    ERC20.transfer.selector, _user, reward
                )
            );
            if (success && (data.length == 0 || abi.decode(data, (bool)))) {
                rewards[_user][_rewardsToken] = 0;
                emit RewardPaid(_user, _rewardsToken, reward);
            } else {
                continue;
            }
        }
    }
}
}

```

While these rewards are not lost and the owner can call `internalizeDonations()` later to vest them, this actually results in a delayed update of the rewards, during which time the user who made the deposit will enjoy the rewards that don't belong to him and the user who made the withdrawal will lose them.

Note that the comment for the `totalAssets()` function mentions that `getRewardForUser()` will convert reward tokens to donated tokens, so this may be known behavior.

```

/// @dev Not yet harvested rewards not yet added due to possible temporal overestimation due to rewards being
↪ donated through `getRewardForUser`
function totalAssets() public view override virtual returns (uint amountInAsset) {

```

But the more serious case where an attacker always calls `getRewardForUser()` to make the rewards in `_harvestRewards()` 0, the token will not be added to `RewardedTokens`:

```

if (rewards == 0) continue; /// @dev Skip if no rewards, saves from iVault revert 'Cannot stake 0'

(rewards, _token) = _autoCompoundHook(_token, _ibgt, _ibgtVault, rewards);

// 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)) {
    uint fee = rewards * _performanceFee / BP;
    uint netRewards = rewards - fee;

    if (_token == asset()) {
        iVault.stake(netRewards);
    }

    _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

```

Since `internalizeDonations()` requires tokens to be `RewardedTokens`. This prevents the owner from calling `internalizeDonations()` to vest those tokens.

```

require(!_isRewardedToken(token), "CollVault: token not rewarded");

```

Recommendation: This issue makes the reward tokens act as donation tokens, and one option is to treat the donation tokens as reward tokens directly, and use the donation tokens and reward tokens to update the rewards in `_harvestRewards()`. If converting reward tokens to donated tokens is the expected behavior, it needs to be ensured that tokens are added to `RewardedTokens` even if `rewards == 0` in `_harvestRewards()`:

```

uint rewards = newBalance - prevBalances[i];
- if (rewards == 0) continue; /// @dev Skip if no rewards, saves from iVault revert 'Cannot stake 0'

(rewards, _token) = _autoCompoundHook(_token, _ibgt, _ibgtVault, rewards);

// 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)) {
    uint fee = rewards * _performanceFee / BP;
    uint netRewards = rewards - fee;

-   if (_token == asset()) {
+   if (_token == asset() && netRewards != 0 ) {
        iVault.stake(netRewards);
    }

    _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

```

Beraborrow: Acknowledged in previous audits. See Issue [M-1](#).

3.1.5 The order of tokens returned by `CollVaultRouter.previewRedeemUnderlying()` is not the same as that used in `redeemToOne()`

Submitted by [cccZ](#)

Severity: Medium Risk

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

Description: *Router contracts provide preview functions to get the redeemed tokens and amounts, in addition to the user can easily preview the redemption results, these results are also convenient for the user to select the appropriate parameters for the token swap in `redeem*ToOne()`. For example, the results of `LSPRouter's previewRedeemPreferredUnderlying()` can be used to prepare the token swap parameters needed for `redeemPreferredUnderlyingToOne()`. The results of `CollVaultRouter's previewRedeemUnderlying()` can be used to prepare the token swap parameters needed for `redeemToOne()`.

However, the problem here is that the order of tokens returned by `previewRedeemUnderlying()` is not the same as the order of tokens needed for `redeemToOne()`, which results in users using the results of `previewRedeemUnderlying()` to execute `redeemToOne()` to fail. `redeemToOne()` will call `tryGetRewardedTokensIncludingIbgtVault()` directly to get the redeemed tokens:

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

while `previewRedeemUnderlying()` will get the redeemed tokens during the simulation:

```
// Recursively simulate if token is a nested vault
if (token == address(ibgtVault) && token != address(vault)) {
    IIrreversibleCollateralVault nestedVault = IIrreversibleCollateralVault(token);
    _simulateVaultRedemption(nestedVault, tokenAmount, tokens, amounts, true);
} else {
    TokenValidationLib.aggregateIfNotExist(token, tokenAmount, tokens, amounts);
}
```

For example, consider `collVault` with asset token A and the reward token is token B. The order of redeemed tokens returned by `previewRedeemUnderlying()` is A, B, and the order of tokens used in `redeemToOne()` will be B, A, so when the user sets the `pathDefinitions` parameter for the token exchange in `redeemToOne()` based on A, B, that parameter will be mismatched, resulting in the execution failure. One proof of this is in the test where it sets the token swap parameter based on the order of tokens returned by `previewRedeemUnderlying()`.

```
function _buildMultiPathSwapsCalldata(
    IIrreversibleCollateralVault collVault,
    uint shares,
    address targetToken
) internal returns (
    bytes[] memory pathDefinitions,
    uint[] memory minOutputs,
    uint[] memory quoteAmounts,
    address[] memory tokens,
    address executor
) {
    (address[] memory _tokens, uint[] memory amounts) = collVaultRouter.previewRedeemUnderlying(collVault,
        ↪ shares);
    tokens = _tokens;

    pathDefinitions = new bytes[](tokens.length);
    minOutputs = new uint[](tokens.length);
    quoteAmounts = new uint[](tokens.length);

    for (uint i; i < tokens.length; i++) {
        // if amount lower than 1e3 dont try to claim as ob api reverts prob to rounding errs
        if (tokens[i] != targetToken && amounts[i] > 1e3) {
            SwapInfo memory info = _getSwapInfo(tokens[i], amounts[i], targetToken, address(owner));
            minOutputs[i] = info.outputMin * 25 / 100;
            quoteAmounts[i] = info.outputQuote;
            pathDefinitions[i] = info.pathDefinition;
            executor = info.executor;
        }
    }
}
```

In addition, `LSPRouter._previewWithdrawUnderlyingCollVaultAssets()` also uses the result of `CollVaultRouter.previewRedeemUnderlying()`, that is, the token order returned by `LSPRouter.previewRedeemPreferredUnderlying()` will be A, B, while the token order used in `LSPRouter._withdrawUnderlyingCollVaultAssets()` is B, A.

Recommendation: Since the token order returned by `CollVaultRouter.previewRedeemUnderlying()` is inconsistent with the token order used by `LSPRouter` and `CollVaultRouter`, it is recommended to change the token order returned by `CollVaultRouter.previewRedeemUnderlying()`.

Beraborrow: Full new `_simulateVaultRedemption` inside `previewRedeemUnderlying`, see [the contract diff](#). This implementation was recently audited by the auditor Santipuu.

3.1.6 CollVaultRouter::previewRedeemUnderlying is miscalculating all iBGT unclaimed rewards

Submitted by [santipu](#)

Severity: Medium Risk

Context: CollVaultRouter.sol#L307-L312

Summary: The function `previewRedeemUnderlying` miscalculates all unclaimed iBGT rewards, leading to the resulting amount being lower than expected, causing users to lose funds due to slippage on `redeemToOne` and `redeemPreferredUnderlyingToOne`.

Finding Description: The function `previewRedeemUnderlying` within `CollVaultRouter` is used to simulate a redemption from a Collateral Vault and get the tokens and amounts that will be received upon the redemption. The actual redemption simulation occurs on the internal function `_simulateVaultRedemption`, which is called from `previewRedeemUnderlying`. However, the calculation of the amounts to be received is flawed as it does not take into account all unclaimed iBGT rewards.

```
function _simulateVaultRedemption(
    IInfraredCollateralVault vault,
    uint sharesToRedeem,
    DynamicArrayLib.DynamicArray memory tokens,
    DynamicArrayLib.DynamicArray memory amounts,
    bool isNested
) internal view {
    // ...

    address[] memory allTokens = vault.tryGetRewardedTokens();

    for (uint256 i; i < allTokens.length; i++) {
        // ...

        try vault.infraredVault() returns (IInfraredVault infraredVault) {
            v.earned = infraredVault.earned(token, address(vault)); // <<<
            if (token == address(ibgtVault)) {
                // Internal earned amount not included, returned amount will probably be slightly lower
                v.earned = ibgtVault.previewDeposit(v.earned); // <<<
            }
        } catch {
            v.earned = 0;
        }
        // ...
    }
}
```

The function has to calculate the unclaimed rewards from the underlying Infrared vault (i.e. `v.earned`) to later add them to the total rewards and return the amount that belongs to the user simulating the redemption.

However, the issue is that when the reward token is `ibgtVault`, the resulting unclaimed rewards will always be zero, leading to an inaccurate reward amount that will be lower than it should be. That is because when the reward token is `ibgtVault`, the underlying reward token on the Infrared vault will always be `iBGT`, and not `ibgtVault`. Therefore, when trying to calculate the `earned` amount on Infrared Vault of the token `ibgtVault`, the resulting amount will be zero, and the unclaimed iBGT rewards won't be taken into account.

Impact Explanation: The function `previewRedeemUnderlying` is vital for two reasons:

1. It's necessary to know which will be the tokens and amounts received upon redeeming from a Collateral Vault, so we can accurately swap them through `CollVaultRouter::redeemToOne`.
2. The affected function `previewRedeemUnderlying` is also called from `LSPRouter::previewRedeemPreferredUnderlyingToOne`, which is also vital for later calling `redeemPreferredUnderlyingToOne`.

In summary, both functions `redeemToOne` and `redeemPreferredUnderlyingToOne` directly depend on `previewRedeemUnderlying` to know the tokens and amounts that will be available to swap after redeeming from a Collateral Vault.

Given that the unclaimed iBGT rewards won't be taken into account due to this issue, users will believe that they will receive fewer tokens than expected, leading to having lower slippage values, and lose these extra tokens due to getting sandwiched on the swaps.

Additionally, all Boyco Vaults use the function `redeemToOne` when the following functions are called: `adjustBoycoDen`, `closeBoycoDen`, `unwrapCollVaultToAsset`, and `claimCollateral`. Given that currently, all deployed Boyco Vaults have a TVL of more than 400M, when the owner calls any of the mentioned functions it will lead to the loss of some funds given the incorrect arguments that are set from calling the flawed `previewRedeemUnderlying` function.

Likelihood Explanation: This issue will only occur when there are some pending unclaimed iBGT rewards on the Infrared vault, which can only happen when the Collateral Vault is not called during some blocks. Considering that Beraborrow will have whitelisted more than 20 Collateral Vaults, and that ALL of them distribute iBGT as the main reward token, it's safe to assume that some of these vaults may not be so active, leading to these unclaimed rewards to being accounted for in the redeeming simulation, and causing the loss of funds for users.

Recommendation: To mitigate this issue, is recommended to properly account for these unclaimed iBGT rewards:

```
for (uint256 i; i < allTokens.length; i++) {
    address token = allTokens[i];

    // Skip if it's the primary asset
    if (token == v.asset) continue;

    try vault.infraredVault() returns (IIInfraredVault infraredVault) {
-       v.earned = infraredVault.earned(token, address(vault));
        if (token == address(ibgtVault)) {
            // Internal earned amount not included, returned amount will probably be slightly lower
+       v.earned = infraredVault.earned(ibGT, address(vault));
        v.earned = ibgtVault.previewDeposit(v.earned);
-    }
+    } else {
+       v.earned = infraredVault.earned(token, address(vault));
+    }
    } catch {
        v.earned = 0;
    }
    // ...
}
```

Beraborrow: Fixed, see [the contract diff](#).

3.1.7 All unclaimed rewards are not being factored in `previewRedeemUnderlying` due to wrong argument ordering

Submitted by [santipu](#)

Severity: Medium Risk

Context: `CollVaultRouter.sol#L288`, `CollVaultRouter.sol#L308`

Description: The function `previewRedeemUnderlying` does not take into account the unclaimed rewards on the underlying Infrared Vault due to the wrong ordering of arguments. On `previewRedeemUnderlying`, the `earned` function on Infrared vault is called to calculate the unclaimed amount of rewards:

```
function _simulateVaultRedemption(
    IIInfraredCollateralVault vault,
    uint sharesToRedeem,
    DynamicArrayLib.DynamicArray memory tokens,
    DynamicArrayLib.DynamicArray memory amounts,
    bool isNested
) internal view {
    SimRedeemVars memory v;

    v.netShares = isNested ? sharesToRedeem : sharesToRedeem - sharesToRedeem.feeOnRaw(vault.getWithdrawFee());
    v.totalSupply = vault.totalSupply();
    v.asset = vault.asset();

    // If not an InfraredVault, the try-catch just skips
    try vault.infraredVault() returns (IIInfraredVault infraredVault) {
        v.earned = infraredVault.earned(v.asset, address(vault)); // <<<
    } catch {}

    v.assetAmount = v.netShares.mulDiv(
```

```

        vault.getBalance(v.asset) + v.earned,
        v.totalSupply,
        Math.Rounding.Down
    );

    TokenValidationLib.aggregateIfNotExistent(v.asset, v.assetAmount, tokens, amounts);

    address[] memory allTokens = vault.tryGetRewardedTokens();

    for (uint256 i; i < allTokens.length; i++) {
        address token = allTokens[i];

        // Skip if it's the primary asset
        if (token == v.asset) continue;

        try vault.infraredVault() returns (IInfraredVault infraredVault) {
            v.earned = infraredVault.earned(token, address(vault)); // <<<
            if (token == address(ibgtVault)) {
                // Internal earned amount not included, returned amount will probably be slightly lower
                v.earned = ibgtVault.previewDeposit(v.earned);
            }
        } 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;

        // Recursively simulate if token is a nested vault
        if (token == address(ibgtVault) && token != address(vault)) {
            IInfraredCollateralVault nestedVault = IInfraredCollateralVault(token);
            _simulateVaultRedemption(nestedVault, tokenAmount, tokens, amounts, true);
        } else {
            TokenValidationLib.aggregateIfNotExistent(token, tokenAmount, tokens, amounts);
        }
    }
}

```

However, the arguments to call `earned` are not ordered correctly, which will cause the unclaimed rewards to be zero. According to the [Infrared docs](#), the interface for the `earned` function is the following:

```
function earned(address account, address _rewardsToken) public view returns (uint256);
```

But when we call `earned` from `_simulateVaultRedemption`, the arguments are the other way around:

```

v.earned = infraredVault.earned(v.asset, address(vault));

// ...

v.earned = infraredVault.earned(token, address(vault));

```

This discrepancy will cause all unclaimed rewards (i.e. `v.earned`) will always be zero, causing the final amounts to be lower than they should be.

The impact of this issue is the same as in #20, it will cause a loss of funds in the end due to the wrong slippage arguments. However, the root causes of this issue and #20 are essentially different, hence the different reports.

Recommendation: To mitigate this issue, is recommended to switch the arguments on `earned`:

```

- v.earned = infraredVault.earned(v.asset, address(vault));
+ v.earned = infraredVault.earned(address(vault), v.asset);

// ...

- v.earned = infraredVault.earned(token, address(vault));
+ v.earned = infraredVault.earned(address(vault), token);

```

Beraborrow: Fixed, see [the contract diff](#).

3.1.8 NECT is not distributed to validators on ValidatorPool

Submitted by [santipu](#)

Severity: Medium Risk

Context: ValidatorPool.sol#L56-L58

Description: The ValidatorPool contract is used to send some funds on liquidations so that they can be distributed between validators. Whenever a liquidation happens, a portion of the collateral and debt gas compensation is sent to ValidatorPool:

```
function finalizeLiquidation(
    address _liquidator,
    uint256 _debt,
    uint256 _coll,
    uint256 _collSurplus,
    uint256 _debtGasComp,
    uint256 _collGasComp
) external {
    // ...

    // Split collateral and debt compensation between liquidator, sNect gauge and validator pools.
    // Send compensation tokens to liquidator
    ILiquidationManager.LiquidationFeeData memory data =
    ↪ ILiquidationManager(liquidationManager).liquidationsFeeAndRecipients();
    debtToken.returnFromPool(gasPoolAddress, _liquidator, _debtGasComp * data.liquidatorFee /
    ↪ DECIMAL_PRECISION);
    // Send compensation tokens to sNect Gauge
    debtToken.returnFromPool(gasPoolAddress, data.sNectGauge, _debtGasComp * data.sNectGaugeFee /
    ↪ DECIMAL_PRECISION);
    // Send compensation tokens to validator pool
    debtToken.returnFromPool(gasPoolAddress, data.validatorPool, _debtGasComp * data.poolFee /
    ↪ DECIMAL_PRECISION); // <<<

    _sendCollateral(_liquidator, _collGasComp * data.liquidatorFee / DECIMAL_PRECISION);
    _sendCollateral(data.sNectGauge, _collGasComp * data.sNectGaugeFee / DECIMAL_PRECISION);
    _sendCollateral(data.validatorPool, _collGasComp * data.poolFee / DECIMAL_PRECISION); // <<<
}
```

On ValidatorPool, the function `distribute` is used to split and transfer those funds (both collateral and NECT) to the specified validators. However, the `distribute` function is only designed to handle collateral tokens, not NECT. This will cause validators only to receive funds only on the collaterals, not on the NECT sitting at the contract.

```
function distribute() external {
    address[] memory collateralTokens = _liquidStabilityPool // <<<
    .getCollateralTokens();

    // ...

    for (uint i; i < collateralTokens.length; i++) { // <<<
        collateralTokenToValidatorAmount[i] = new uint[](validatorCount);

        uint tokenBalance = IERC20(collateralTokens[i]).balanceOf(address(this));
        if (tokenBalance != 0) {
            for (uint j; j < validatorCount; j++) {
                uint amount = (tokenBalance * memShares[j]) / BASIS_POINT;
                if (amount != 0) {
                    collateralTokenToValidatorAmount[i][j] = amount;
                }
            }
        }
    }

    for (uint i; i < collateralTokens.length; i++) { // <<<
        for (uint j; j < validatorCount; j++) {
            if (collateralTokenToValidatorAmount[i][j] != 0) {
                IERC20(collateralTokens[i]).safeTransfer(memValidators[j],
                ↪ collateralTokenToValidatorAmount[i][j]);
            }
        }
    }
}
```

This issue will cause a loss of funds for validators as they will only receive collaterals and not NECT. I've marked this issue as medium severity instead of high given that the function `recoverLockedSunsettedAssets` allows admins to recover some tokens from the contract that are not collaterals. While the admins theoretically could recover the NECT stuck in the contract and distribute it manually to validators, this is not sustainable mitigation if there are lots of liquidations and the validators change frequently. For this reason, I believe medium severity fits this issue.

Recommendation: To mitigate this issue, it's recommended to distribute NECT along with collaterals in distribute.

Beraborrow: Fixed, see [the contract diff](#).

3.1.9 Malicious users can steal all USDC on the `PermissionlessPSM.sol` for free by utilizing the rounding issue in `previewMint(...)`

Submitted by [GeneralKay](#), also found by [vinicaboy](#) and [hash](#)

Severity: Medium Risk

Context: [PermissionlessPSM.sol#L163-L173](#)

Summary: Malicious users can take advantage of the rounding in the `previewMint(...)` function to recursively mint unlimited NECT token for free then burn those NECT tokens to `withdraw` all the USDC and other stable tokens available on `PermissionlessPSM.sol`. This attack vector can be performed in single transaction.

Finding Description: The major issue is a rounding issue in the `previewMint(...)` function that is called by the `mint(...)` function. Let's take a look at the `mint(...)` function:

```
// File: PermissionlessPSM.sol
function mint(address stable, uint nectAmount, address receiver) public notPaused returns (uint stableAmount) {
    uint nectFee;
    (stableAmount, nectFee) = previewMint(stable, nectAmount); // <<<

    uint cap = mintCap[stable];
    uint _nectMinted = nectMinted[stable] + nectAmount;
    if (_nectMinted > cap) revert PassedMintCap(cap, _nectMinted);

    IERC20(stable).safeTransferFrom(msg.sender, address(this), stableAmount);

    nect.mint(receiver, nectAmount);
    nect.mint(metaBeraborrowCore.feeReceiver(), nectFee);

    nectMinted[stable] = _nectMinted;

    emit Deposit(msg.sender, stable, stableAmount, nectAmount, nectFee);
}
```

The issue here lies in the calculation of the `stableAmount`, which is the amount of `stable` that will be transferred from the user to the `PermissionlessPSM.sol` contract. This `stableAmount` is calculated with the user supplied `nectAmount` parameter of the `mint(...)` function and this `nectAmount` is the attackers weapon. Remember that `wadOffset` is the decimal difference between `nect` token and the `stable` token raised to the power of 10. So for USDC `stable`, `wadOffset` will be `1e12`. since USDC has 6 decimals and `nect` has 18 decimals.

```
uint64 wadOffset = uint64(10 ** (nect.decimals() - stable.decimals()));
```

Since the `wadOffset` for USDC `stable` is `1e12` then `1e11` as the `nectAmount` is enough for the calculation of `stableAmount` to round down to zero for each `mint(...)` call:

```

function previewMint(address stable, uint nectAmount) public view returns (uint stableAmount, uint nectFee) {
    uint64 wadOffset = stables[stable];

    if (wadOffset == 0) revert NotListedToken(stable);

    uint fee = feeHook.calcFee(msg.sender, stable, nectAmount, IFeeHook.Action.DEPOSIT);
    fee = fee == 0 ? DEFAULT_FEE : fee;

    nectFee = nectAmount.mulDiv(fee, BP - fee, Math.Rounding.Up);
    stableAmount = (nectAmount + nectFee) / wadOffset; //@audit rounding down to zero.
}

```

So when stableAmount rounds down to zero then zero USDC is transferred from the user to the contract and 1e11 nectAmount is minted to the user. So here users mint NECT token for free.

```

function mint(address stable, uint nectAmount, address receiver) public notPaused returns (uint stableAmount) {
    uint nectFee;
    (stableAmount, nectFee) = previewMint(stable, nectAmount); // <<<

    uint cap = mintCap[stable];
    uint _nectMinted = nectMinted[stable] + nectAmount;
    if (_nectMinted > cap) revert PassedMintCap(cap, _nectMinted);

    IERC20(stable).safeTransferFrom(msg.sender, address(this), stableAmount); // <<<

    nect.mint(receiver, nectAmount); // <<<
    nect.mint(metaBeraborrowCore.feeReceiver(), nectFee);

    nectMinted[stable] = _nectMinted;

    emit Deposit(msg.sender, stable, stableAmount, nectAmount, nectFee);
}

```

Impact Explanation: High:

- Attacker can steal all stable tokens that has less than 18 decimals and allows zero value transfer for free. This is done by first calling the mint(...) function with a nectAmount that will cause the stableAmount calculation to round down to zero, then call withdraw to withdraw all deposits.
- More Nect than stable coin backing is minted making nect lose its peg and reliability as a stable coin.

Likelihood Explanation: High: Because USDC, USDC.e and GUSD tokens have decimals less than 18 and they allow zero value transfers.

Proof of Concept (if required): To test the free minting of Nect token by any user without deposit to back the minted nect, copy and paste the test_MintFreeNect_Kay() function below to the PermissionlessPSMTest contract in the test/core/PermissionlessPSM.t.sol file and run the command below:

```

forge test --match-path test/core/PermissionlessPSM.t.sol -vv --match-test test_MintFreeNect_Kay

```

```
// File: test/core/PermissionlessPSM.t.sol:PermissionlessPSMTest
function test_MintFreeNect_Key() public {
    uint256 nectAmount = 1e11; // less than wadOffset to cause round down to zero.
    address GeneralKey = makeAddr("GeneralKey");
    address myStableToken = stableTokens[3]; // STABLE4 token with 6 decimals.

    uint256 NectBalanceBefore = nectarToken.balanceOf(GeneralKey);

    //Notice how I did not even approve psm to spend my stable Usdc because I'm sending zero usdc for free 1e15
    ↪ Nect.
    //Also notice how any caller can do this even when you dont have the stable because I did not use
    ↪ startPrank().
    //This free minting of Nect can be increased by doing this mint amount through a loop by multiple attacker
    ↪ contracts.
    for(uint256 i; i < 10_000; i++){
        psm.mint(myStableToken, nectAmount, GeneralKey);
    }

    uint256 NectBalanceAfter = nectarToken.balanceOf(GeneralKey);

    uint256 exploit = NectBalanceAfter - NectBalanceBefore;

    assertEq(exploit, 1e15);
    console2.log("Free Nectar tokens minted: ", exploit);
}
```

Exploit Scenario:

1. After unsuspecting users have deposited let's say 10,000,000 USDC on PermissionlessPSM.sol.
2. Bob calls the mint(...) function of PermissionlessPSM.sol with 1e11 as the nectAmount parameter.
3. Due to the rounding in previewMint(...), stableAmount is rounded to zero, so zero USDC is transferred from Bob and 1e11 NECT is minted to Bob.
4. Bob Repeats steps 2 and 3 to get unlimited NECT token for free.
5. Bob uses these minted NECT tokens to withdraw all other user deposits of the 10,000,000.

Recommendation: Consider reverting the transaction when `stableAmount = (nectAmount + nectFee) / wadOffset` rounds to Zero.

3.1.10 Ordered liquidations are always broken due to BrimeDen

Submitted by [santipu](#), also found by [hash](#)

Severity: Medium Risk

Context: [LiquidationManager.sol#L285](#)

Summary: Due to the existence of BrimeDen, ordered liquidations that are executed through the function `liquidateDens` will never work.

Finding Description: The objective of ordered liquidations is to liquidate Dens in order from lower ICR to higher. However, the existence of a special Den with a lower MCR than usual will break this flow and will cause a permanent DoS on `liquidateDens`. The flow of `liquidateDens` is the following:

1. Starting with the Den with lower ICR, loop over all Dens and try to liquidate them until we arrive at the first one that is not liquidatable.
2. When the first loop is over, the second loop also starts with the Den with lower ICR and tries to liquidate with a cap if we're in Recovery Mode. If the system is currently not in RM, this second loop will never execute.

The main issue with this function is that is based on the assumption that all Dens with an ICR below MCR are, *by definition*, liquidatable. However, this assumption is broken due to the existence of BrimeDen.

According to the [Beraborrow audit docs](#), the objective of BrimeDen is to mint NECT more efficiently and absorb redemptions by having an MCR lower than usual, which will allow to have an ICR lower than the usual MCR. This key feature of BrimeDen will break all ordered liquidations.

Example scenario:

1. We have 3 Dens:

- BrimeDen with 106% ICR (BrimeMCR is 105% so it's not liquidatable).
- Den1 with 113% ICR (normal MCR is 120%, so the Den IS liquidatable).
- Den2 with an ICR way higher (not liquidatable).

2. A liquidator calls `liquidateDens` to execute an ordered liquidation:

- The first loop encounters BrimeDen and breaks the loop as it's not liquidatable.
- The second loop is not executed because we're not in Recovery Mode.
- The call reverts as no Dens have been liquidated.

Impact Explanation: Due to the existence of BrimeDen, all ordered liquidations are permanently DoSed, which will bring problems to the liquidator partners. However, the existence of `batchLiquidateDens` allows liquidators to liquidate specific Dens in an unordered manner, which will keep the health of the system. Given that only ordered liquidations are broken instead of ALL liquidations, the severity is marked as medium severity instead of high.

Likelihood Explanation: This issue will occur on ALL ordered liquidations, so the likelihood is high.

Proof of Concept: Run the current test on `LiquidationManager.t.sol`:

```
import {IFactory} from "../../src/interfaces/core/IFactory.sol";

function test_broken_ordered_liquidations_brimeDen() public {
    // Set interest rate and borrowing fees to 0 for simplicity
    IFactory.DeploymentParams memory params = IFactory.DeploymentParams({
        minuteDecayFactor: 999037758833783000, // same as before
        redemptionFeeFloor: 1e18 / 1000 * 5, // same as before
        maxRedemptionFee: 1e18, // same as before
        borrowingFeeFloor: 0, // 0% borrowing fee floor
        maxBorrowingFee: 0, // 0% borrowing fee ceiling
        interestRateInBps: 0, // 0% interest rate
        maxDebt: 6e6 * 1e18, // same as before
        MCR: 1.2e18, // same as before
        collVaultRouter: address(0)
    });
    vm.prank(owner);
    denManager.setParameters(params);

    // Open Den1 with ICR 160%
    _openDen(random);

    // Open Den1 with a higher ICR
    _openDen(depositor);
    _addCollateralToDen(100e18);

    // Open BrimeDen with ICR 150%
    uint collateralAmount = 1.5e18;
    uint debtAmount = 1e18;
    deal(wBERA, owner, collateralAmount);
    bytes memory approveWBERAData = abi.encodeWithSelector(IEC20.approve.selector,
    ↪ address(borrowerOperations), collateralAmount);
    bytes memory multicallData = abi.encodeWithSelector(
        borrowerOperations.openDen.selector,
        denManager,
        address(brimeDen),
        0.1e18, // 10% for max fee
        collateralAmount,
        debtAmount,
        address(0),
        address(0)
    );
    vm.startPrank(owner);
    BrimeDen.Call3[] memory multicallDataArray = new BrimeDen.Call3[](2);
    multicallDataArray[0] = BrimeDen.Call3(wBERA, false, approveWBERAData);
    multicallDataArray[1] = BrimeDen.Call3(address(borrowerOperations), false, multicallData);
    IERC20(wBERA).transfer(address(brimeDen), collateralAmount);
    brimeDen.multicall3(multicallDataArray);
    vm.stopPrank();

    // Collateral price drops from 1e18 to 0.71e18
```

```

vm.warp(block.timestamp + 12 seconds);
uint256 newPrice = 0.71e18;
_mockPriceFeed(newPrice, block.timestamp, 2);

// Check ICRs of all Dens
assertEq(denManager.getCurrentICR(address(brimeDen), newPrice), 1.065e18); // BrimeDen has 106.5% ICR,
↳ it should NOT be liquidated due to special MCR
assertEq(denManager.getCurrentICR(random, newPrice), 1.136e18); // Den1 has 113.6% ICR, it
↳ should be liquidated
assertEq(denManager.getCurrentICR(depositor, newPrice), 72.136e18); // Den2 has a way higher
↳ ICR, it should NOT be liquidated

// Try ordered liquidation
// The call reverts because no Dens are liquidated, even though Den1 should be.
vm.expectRevert("DenManager: nothing to liquidate");
liquidationManager.liquidateDens(denManager, 10, 100e18, liquidator);
}

```

Recommendation: To mitigate this issue, I see two possible options:

1. Remove the existence of BrimeDen so that all liquidation mechanisms still work correctly.
2. Remove the function `liquidateDens` as it will never work alongside BrimeDen.

Beraborrow: Agree on the issue. We won't open any BrimeDen for longer than a few amount of hours, either to fix 'Rounding can cause sum of individual den debts to be greater than totalActiveDebt causing underflow on withdrawals' issue, or temporally stop incoming redemptions. Since we only plan to have it opened for very short period of times, the likelihood of the impact lowers abruptly, since it takes time for other dens to accrue further interest, and also liquidations timing would be unlikely.

3.1.11 Incorrect `nectMinted` calculation in `pPermissionlessPSM`

Submitted by [cccz](#), also found by [GeneralKay](#) and [hash](#)

Severity: Medium Risk

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

Description: In `PermissionlessPSM`, users can provide stablecoins to mint NECT, the amount of NECT minted is recorded by `nectMinted`, and `nectMinted` cannot exceed `mintCap`:

```

function deposit(address stable, uint stableAmount, address receiver) public notPaused returns (uint
↳ mintedNect) {
    uint nectFee;
    (mintedNect, nectFee) = previewDeposit(stable, stableAmount);

    uint cap = mintCap[stable];
    uint _nectMinted = nectMinted[stable] + mintedNect;
    if (_nectMinted > cap) revert PassedMintCap(cap, _nectMinted);

    IERC20(stable).safeTransferFrom(msg.sender, address(this), stableAmount);

    nect.mint(receiver, mintedNect);
    nect.mint(metaBeraborrowCore.feeReceiver(), nectFee);

    nectMinted[stable] = _nectMinted;

    emit Deposit(msg.sender, stable, stableAmount, mintedNect, nectFee);
}

```

However, the problem here is that the actual amount of NECT minted is greater than the value recorded by `nectMinted`. For example, consider that the fee is 1% and the user provides 1000 USDC, 990 NECTs are minted to the user and the `nectMinted` is increased by 990, however 10 NECTs of the fee are minted to the `feeReceiver`, which results in 1000 NECTs being minted instead of 990.

The NECTs in the `feeReceiver` can be taken out and traded in the market.

```

contract FeeReceiver is BeraborrowOwnable {
    using SafeERC20 for IERC20;

    constructor(address _beraborrowCore) BeraborrowOwnable(_beraborrowCore) {}

    function transferToken(IERC20 token, address receiver, uint256 amount) external onlyOwner {
        token.safeTransfer(receiver, amount);
    }

    function setTokenApproval(IERC20 token, address spender, uint256 amount) external onlyOwner {
        token.safeIncreaseAllowance(spender, amount);
    }
}

```

When redeemed, the NECT amount to be redeemed is subtracted from nectMinted:

```

function redeem(address stable, uint nectAmount, address receiver)
    public
    notPaused
    returns (uint stableAmount)
{
    uint stableFee;
    (stableAmount, stableFee) = previewRedeem(stable, nectAmount);

    nect.burn(msg.sender, nectAmount);

    nectMinted[stable] -= nectAmount;

    IERC20(stable).safeTransfer(receiver, stableAmount);
    IERC20(stable).safeTransfer(metaBeraborrowCore.feeReceiver(), stableFee);

    emit Withdraw(msg.sender, stable, stableAmount, nectAmount, stableFee);
}

```

So in the above example, the user can burn 990 NECTs to redeem 990 USDC, but when the feeReceiver's NECTs are redeemed, the nectMinted is already 0, and the redeem will fail due to underflow, resulting in the feeReceiver's NECTs not being redeemed even though there is sufficient USDC in the PermissionlessPSM.

Recommendation: It is recommended to consider nectFee when increasing nectMinted.

```

    uint cap = mintCap[stable];
    - uint _nectMinted = nectMinted[stable] + nectAmount;
    + uint _nectMinted = nectMinted[stable] + nectAmount + nectFee;
    if (_nectMinted > cap) revert PassedMintCap(cap, _nectMinted);

    IERC20(stable).safeTransferFrom(msg.sender, address(this), stableAmount);

    nect.mint(receiver, nectAmount);
    nect.mint(metaBeraborrowCore.feeReceiver(), nectFee);

    nectMinted[stable] = _nectMinted;

```

Beraborrow: Fixed in line 109 at contract [0xB2F796FA30A8512C1D27a1853a9a1a8056b5CC25](#).

3.1.12 Sunsetting collateral will lower TCR and trigger recovery mode

Submitted by [santipu](#), also found by [hash](#)

Severity: Medium Risk

Context: [BorrowerOperations.sol#L614-L628](#)

Description: When a collateral being used on Beraborrow becomes unstable and starts dropping its price heavily, the owners can sunset that collateral to prevent any new debt being taken with that token, limiting the risk. However, this safeguard is limited as a sunsetted collateral will still influence the TCR until all debt has been cleared (either redeemed, closed, or liquidated). Taking into account sunsetted collateral in the TCR calculation is an unnecessary risk, as it will significantly lower the TCR whenever a specific collateral becomes unstable and too volatile.

Example scenario:

1. Beraborrow has 3 active collaterals (e.g. Coll1, Coll2 and Coll3).
2. After some time, the protocol behind issuing Coll3 is hacked, resulting in the price of Coll3 dropping significantly given that its underlying value has been stolen.
3. The admins of Beraborrow see that and decide to sunset Coll3 to limit risks.
4. The price of Coll3 keeps dropping, causing the ICR of its Dens to lower, dragging the whole TCR down, until Recovery Mode is triggered.
5. When RM is triggered, a lot of Dens with $ICR > MCR$ will end up liquidated and losing up to 0.5% of their collateral value.

Even though Recovery Mode was designed to handle situations like this, it's unnecessary to trigger it due to a sunsetted collateral, as it will cause the loss of funds for some users. In a scenario like this, we cannot expect a lot of redemptions happening on Coll3, as that token has become undesirable and no users will want it. It's even possible that bots refuse to take it due to its market price (e.g. on uniswap) being lower than the oracle reported price.

Given that Beraborrow plans to add around 20 collaterals, it's reasonable to assume that over time some of them will be sunsetted due to adverse market conditions and a falling price. In those situations, the TCR should not be lowered due to this sunsetted collateral.

Recommendation: To mitigate this issue, it's recommended to modify the TCR calculations on `Borrower-Operations` (`_getTCRData`) and not take into account the aggregated ICR of sunsetted collateral tokens.

Beraborrow: The auditor claims "taking into account sunsetted collateral in the TCR calculation is an unnecessary risk, as it will significantly lower the TCR whenever a specific collateral becomes unstable and too volatile".

If we are about to fall into RecoveryMode because of such a faulty collateral, even if it's sunsetted, it RM should be applicable, since we care about NECT peg, and offsetting this debt with the Stability Pool should be the main priority to keep NECTs overcollateralization.

The auditor also claims "Even though Recovery Mode was designed to handle situations like this, it's unnecessary to trigger it due to a sunsetted collateral, as it will cause the loss of funds for some users".

A sunsetted collateral still can depeg NECT, which would also cause loss of funds for all other users.

3.1.13 Rounding can cause sum of individual den debts to be greater than `totalActiveDebt` causing underflow on withdrawals

Submitted by hash

Severity: Medium Risk

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

Description: The total debt interest calculation can suffer more precision losses than the individual Den's. This can cause the sum of individual den's debt to be greater than the total interest added to `totalActiveDebt` disallowing the final withdrawals because of underflow:


```

vm.stopPrank();

vm.warp(block.timestamp + time1);
vm.prank(address(liquidationManager));
denManager.updateBalances();
vm.warp(block.timestamp + time2);
currentTotalDebt = denManager.getTotalActiveDebt();
(uint256 debt, uint256 coll,,) = denManager.getEntireDebtAndColl(depositor);

int256 diff = int256(debt) - int256(currentTotalDebt);
console2.log("diff", diff);

// assert(debt <= currentTotalDebt);

vm.startPrank(depositor);
deal(address(nectarToken), depositor, debt);
borrowerOperations.closeDen(denManager, depositor);
vm.stopPrank();
}

```

Recommendation: When subtracting take the min of totalActiveDebt and debtToSubtract.

Beraborrow: There will always be a fixed protocol position, so even if we desire to withdraw that position, we can effectuate the BrimeDen deposit to net out the interest caused underflow (see the [comment in cantina code](#)).

3.1.14 Liquidation of protocol den will cause locked funds for boyco den

Submitted by [hash](#)

Severity: Medium Risk

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

Description: Boyco den deposits assets into a den manager alongside protocol den:

```

contract PermissionedDenManager is DenManager {
    address public permissionedDen;
    address public protocolDen;

    function _isPermissionedCheck(address _borrower) internal view override {
        require(_borrower == permissionedDen || _borrower == protocolDen, "PermissionedDenManager: Only the
        → Permissioned/ProtocolDen can open a position");
    }
}

```

When closing the boyco den, only $(1 - \text{nectOutThreshold})$ can be externally donated by the owner without suffering a loss. In case this amount is not enough, the den cannot be closed. This is based on the assumption that the initially obtained nect amount would be enough to cover the current debt mostly:

```

function closeBoycoDen(PartialRedeemToOneParams calldata collRouterParams)
    external
    onlyOwner
    isBoycoDenOpened
    isAfterPromotion
{
    // ...

    uint256 nectBalance = nect.balanceOf(address(this));
    (uint256 denColl, uint256 denDebt) = denManager.getDenCollAndDebt(address(this));
    // CloseDen requires to burn denDebt - DEBT_GAS_COMPENSATION
    denDebt -= $.borrowerOperations.DEBT_GAS_COMPENSATION();

    if (nectBalance < denDebt) {
        uint256 nectOutThreshold = $.nectOutThreshold;
        uint256 leftOverToRepaymentInBP = nectBalance * BP / denDebt;

        if (leftOverToRepaymentInBP < nectOutThreshold) { // <<<
            revert NectThresholdTooLow(leftOverToRepaymentInBP, nectOutThreshold);
        }
    }
}

```

This is problematic as if the protocolDen gets liquidated, its debt will also be added to the boyco den making this amount insufficient. Now the boyco den cannot be closed.

Recommendation: Making a setter function for `nextOutThreshold` will allow the admin to adjust the limit but there could still be issues because if the amount is large, then the paid out extra amount could result in underflow.

Beraborrow: Won't be fixing since we don't consider the setter to realistically change anything. On top of that, we consider it not applicable since the `protocolDen` won't be ever liquidated in a realistic scenario. Totally agree with [Santipus comment](#).