SHERLOCK

Security Review For Beraborrow

Collaborative Audit Prepared For:	**Beraborrow**
Lead Security Expert(s):	**GalloDaSballo** **hyh** **santipu_**
Date Audited:	**November 20 - December 27, 2024**
Final Commit:	**bd458a8**

Introduction

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

Scope

Repository: Beraborrowofficial/blockend

Audited Commit: 89ba557bbde708196fc9b54ab06e86bdce1b64b0

Final Commit: bd458a8bca51e9f493f8d50c0c114c5553817d8b

Final Commit Hash

bd458a8bca51e9f493f8d50c0c114c5553817d8b

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
9	27	41

Issues Fixed or Acknowledged

High	Medium	Low/Info
9	27	41

Issue H-1: Pyth Allows 2 prices per block which allows for Risk Free Triggering of Recovery Mode to liquidate victims at no risk to the attacker

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/161

The protocol has acknowledged this issue.

Summary

The Pyth wrapper will return the latest price from Pyth, instead of the latest price that was updated via the Wrapper:
https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/periphery/PythWrapper.sol#L137-L171

```
/**
     * @dev To wrap Pyth to Chainlink we have to mock roundIds, hence this function
↪    mutates state and is not staticallable
     * @dev Pyths `getPrice` does staleness checks by default
     * @dev Returned roundId is not necessarily the same price as the stored in
↪    roundIdToPrice
     */
    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        uint80 currRoundId = currentRoundId;
        PythStructs.Price memory price = pyth.getPrice(priceId);

        _validation(price);

        // If the price has not yet been updated, we return the previous round
        // The 'roundId == feedIdUpdatesSinceKeeperEnabled' invariant breaks when
↪    the keeper is not fast enough to `storeNewPrice` and price is updated again by
↪    Pyth
        if (price.publishTime != roundIdToPrice[currRoundId].publishTime) {
            ++currRoundId;
```

```
        }

        return (
            currRoundId,
            int256(price.price),
            price.publishTime,
            price.publishTime,
            currRoundId
        );
    }
```

Due to this, anytime Pyth has a Price (A) that is higher than a second price (B)

Then an attacker will be able to sidestep the Recovery Mode threshold check by:

- Opening up a position that drives the TCR down to be >= to CCR
- Update the price of their position to trigger RM

This will allow them to liquidate victims in the same transaction at no risk to them

Vulnerability Detail

The following check is present in `BorrowerOperations`

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/core/BorrowerOperations.sol#L227-L241

```
if (isRecoveryMode) {
    _requireICRisAboveCCR(vars.ICR);
} else {
    _requireICRisAboveMCR(vars.ICR, denManager.MCR(), account);
    uint256 newTCR = _getNewTCRFromDenChange(
        vars.totalPricedCollateral,
        vars.totalDebt,
        _collateralAmount * vars.price,
        true,
        vars.compositeDebt,
        true
    ); // bools: coll increase, debt increase
    _requireNewTCRisAboveCCR(newTCR);
}
```

Which is meant to ensure that no one can trigger RM

The following logic is present in the `PythWrapper`
https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/periphery/PythWrapper.sol#L63-L80

```
function storeNewPrice() public {
    // Enforce `storeNewPrice` call is the only interaction of the transaction
    // Prevents this to be called in an atomical arbitrage transaction that could
↪   by pass PriceFeed deviation checks
    require(msg.sender == tx.origin, "Caller must be EOA");
    // Prevent 2 updates in the same block
    require(block.number > latestBlockNumber, "PythWrapper: block already updated");

    PythStructs.Price memory price = pyth.getPrice(priceId);

    _validation(price);

    if (price.publishTime != roundIdToPrice[currentRoundId].publishTime) {
        ++currentRoundId;
    }

    roundIdToPrice[currentRoundId] = price;
    latestBlockNumber = uint176(block.number);
}
```

Which attempts to prevent updating the price multiple times in a block

The same protection is not present in `latestRoundData()` which will be consumed by the `PriceFeed`

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/periphery/PythWrapper.sol#L142-L171

By combining these two ideas we can trigger RM and liquidate victims

Impact

The impact is that no borrower will ever be able to borrow below the CCR whenever a Pyth Feed is used

Code Snippet

The liquidation manager will go through this logic and perform a capped liquidation

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/core/LiquidationManager.sol#L498-L510

```
} else {
    if (denManagerValues.sunsetting) continue;
    uint256 TCR = BeraborrowMath._computeCR(entireSystemColl, entireSystemDebt);
    if (TCR >= CCR || ICR >= TCR) continue;
    singleLiquidation = _tryLiquidateWithCap(
        denManager,
```

```
        account,
        debtInStabPool,
        denManagerValues.MCR,
        denManagerValues.price
    );
    if (singleLiquidation.debtToOffset == 0) continue;
}
```

Tool used

Manual Review

Recommendation

I believe you can reduce the impact by ensuring that the same price is consumed on each block

This will change the attack from guaranteed to statistical

I believe the statistical attack poses too high of a risk as well, meaning I believe Pyth is not compatible with the Liquity Model, unless you find a way to prevent spam updates or you rethink the logic for Recovery Mode

Discussion

alex-beraborrow

The storing of the prices is only used so that we can make a deviation check in `PriceFeed::\_isPriceChangeAboveMaxDeviation`, `storeNewPrice` has no direct effect into RecoveryMode calculations, the only thing that can cause is a revert in `\_isPriceChangeAboveMaxDeviation`-

GalloDaSballo

```
_isPriceChangeAboveMaxDeviation
```

Unfortunately if that's the case then Pyth opens up to this vector as ultimately anytime you can get 2 prices in a row, where B < A, you can simply use A to bring the system to CCR and then push the price B to have it trigger Recovery mode and liquidate at least one victim Den

alex-beraborrow

If I understand correctly, the attack vector is if the user sees that he can pulls a price that is lower (B) than the current (A), he can grief by withdrawing from an existent position (lowering TCR enough...) so that when he realizes price B the system falls into RM, liquidating one position.

Some questions:

1. How is this attack vector specific to `PythWrapper` and different to a push oracle? This scenario could be done frontrunning an incoming a push oracle update if B < A.
2. Other than griefing, which incentive does the user have to do such a thing? Even if he would manage to atomically secure himself a liquidation, gaining `Gas Compensation` (liquidation reserve + 0.5% of collateral). The borrowing fees he would have to pay to be able to move TCR are significant.

GalloDaSballo

> If I understand correctly, the attack vector is if the user sees that he can pulls a price that is lower (B) than the current (A), he can grief by withdrawing from an existent position (lowering TCR enough...) so that when he realizes price B the system falls into RM, liquidating one position.
> Some questions:
> 1. How is this attack vector specific to `PythWrapper` and different to a push oracle? This scenario could be done frontrunning an incoming a push oracle update if B < A.
> 2. Other than griefing, which incentive does the user have to do such a thing? Even if he would manage to atomically secure himself a liquidation, gaining `Gas Compensation` (liquidation reserve + 0.5% of collateral). The borrowing fees he would have to pay to be able to move TCR are significant.

Great questions, yes the attack is based on the fact that we can open a position with price A, and then push an update and trigger Recovery mode with price B So I agree with you that the issue is not strictly tied to the PythWrapper but to using a push oracle like Pyth in general I wrote about PythWrapper because I thought the code in it was meant to prevent this attack, however it is more dependent on using Pyth in general

In terms of gains and costs, I agree that not all liquidations can be profitable per the borrowing fees, however an attack can simply pick the price changes that suit them the most

Unless some changes was made to LiquidationLibrary, an attack can also pick the targets to liquidate, to maximize their profit, for example they could liquidate the safest Dens first, with the goal of being able to liquidate more total collateral than a more bening liquidator would

alex-beraborrow

> So I agree with you that the issue is not strictly tied to the PythWrapper but to using a push oracle like Pyth in general

Correct, and the same can happen with pull.

> In terms of gains and costs, I agree that not all liquidations can be profitable per the borrowing fees, however an attack can simply pick the price changes that suit them the most
> Unless some changes was made to LiquidationLibrary, an attack can also pick the targets to liquidate, to maximize their profit, for example they could liquidate the safest Dens first, with the goal of being able to liquidate more total collateral than a more benign liquidator would.

Correct, it's something that I don't like about the `LiquidationManager`, the users that should be liquidated in `Recovery Mode` should be the in order the lowest dens, not any arbitrary one that happens to be under `TCR`. On top of what you said, not only they could liquidate the safest one, but the biggest one under TCR, maximizing their 0.5% collateral profits. This is a change we could probably apply after this audit @a-melnichuk.

@GalloDaSballo On top of the `LiquidationManager` change, which other one would you recommend?

GalloDaSballo

> > So I agree with you that the issue is not strictly tied to the PythWrapper but to using a push oracle like Pyth in general
>
> Correct, and the same can happen with pull.
>
> > In terms of gains and costs, I agree that not all liquidations can be profitable per the borrowing fees, however an attack can simply pick the price changes that suit them the most Unless some changes was made to LiquidationLibrary, an attack can also pick the targets to liquidate, to maximize their profit, for example they could liquidate the safest Dens first, with the goal of being able to liquidate more total collateral than a more benign liquidator would.
>
> Correct, it's something that I don't like about the `LiquidationManager`, the users that should be liquidated in `Recovery Mode` should be the in order the lowest dens, not any arbitrary one that happens to be under `TCR`. On top of what you said, not only they could liquidate the safest one, but the biggest one under TCR, maximizing their 0.5% collateral profits. This is a change we could probably apply after this audit @a-melnichuk.
> @GalloDaSballo On top of the `LiquidationManager` change, which other one would you recommend?

The biggest change I personally suggested was to add a small delay before liquidations in Recovery Mode can be performed

This is not a super easy change to make, but you can see how we did it for eBTC a year ago: https://github.com/ebtc-protocol/ebtc/blob/c9b95ac66b4d9093298232b47d93bcaa212e5e0d/packages/contracts/contracts/CdpManagerStorage.sol allowbreak #L27-L115

Fundamentally a brief (15 minutes / 30 minutes) delay, ensures that an attacker is also at risk And since the attacker will most likely have a big position, they would be at the mercy of the MEV auction This massively reduces the incentives to do the attack

alex-beraborrow

I really like the idea. Will have to internally discuss it, but I see the positive asymmetric trade-offs that both removing LiquidationManager custom den liquidations on RM and the grace period offers.

a-melnichuk

Good find @GalloDaSballo. Here's what I'm thinking:

- Make `storeNewPrice` owned. Only keeper can call it. The problem doesn't disappear but now the attacker would have to highjack keeper's private keys.
- Push each price update into an array. Use the latest price only if it wasn't updated in the same block. Use the previous price otherwise (given it's fresh enough etc.)

GalloDaSballo

> Good find @GalloDaSballo. Here's what I'm thinking:
> - Make `storeNewPrice` owned. Only keeper can call it. The problem doesn't disappear but now the attacker would have to highjack keeper's private keys.
> - Push each price update into an array. Use the latest price only if it wasn't updated in the same block. Use the previous price otherwise (given it's fresh enough etc.)

I think if you're willing to maintain the Pyth Updates, then the best next steps are to:

- Consider using Pyth exactly like a pull oracle, meaning it updates every X time at Y Deviation
- Go through historical Pyth data to verify if this is acceptable
- Create PythCLFeed that follows these patterns

This brings you back to a more well known attack area where technically an Oracle Update could be sandwiched, but the likelihood of this is crazy small compared to a risk free attack

GalloDaSballo

The Issue was not mitigated in my opinion, I shared a private gist with the team

Issue H-2: `CollVaultRouter.redeemToOne` allows caller to pass in arbitrary user, allowing them to redeem vault tokens owned by other users

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/162

Summary

`redeemToOne` allows passing `RedeemToOneParams calldata params` which is used, without sanitization, as follows:

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

Vulnerability Detail

The arbitrary `params.owner` can be different from the `msg.sender`

Allowing anyone to steal vault tokens from users that granted approval to the `CollVaultRouter`

Impact

Total loss of funds for users that granted approval to the router

Code Snippet

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/periphery/CollVaultRouter.sol#L515-L528

```
function redeemToOne(
    RedeemToOneParams calldata params
) external {
    address[] memory tokens = params.collVault.rewardedTokens();
    (address[] memory rewardTokens, uint length) =
↪   tokens.underlyingCollVaultAssets(params.collVault.asset());
    uint[] memory prevBalances = new uint[](length);

    for (uint i; i < length; i++) {
        prevBalances[i] = IERC20(rewardTokens[i]).balanceOf(address(this));
    }
```

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

uint prevTargetTokenBalance =
↪ IERC20(params.targetToken).balanceOf(params.receiver);
```

Tool used

Manual Review

Recommendation

The `params.owner` should be changed to `msg.sender`

Discussion

alex-beraborrow

I think you meant `redeemToOne`, not `depositFromAny`

GalloDaSballo

You're right, will fix shortly

On Thu, 28 Nov 2024 at 20:20, alex-beraborrow @.***> wrote:

> I think you meant redeemToOne, not depositFromAny
> – Reply to this email directly, view it on GitHub https://github.com/sherlock-audit/2024-11-beraborrow/issues/162
> allowbreak #issuecomment-2506113854, or unsubscribe https://github.com/notifications/unsubscribe-auth/ADGDQZWO5PEG6UTE3YV2TXT2C4J77AVCNFSM6AAAAABSVCFBIWVHI2DSMVQWIX3LMV43OSLTON2WKQ3PNVWWK3TUHMZDKMBWGEYTGOBVGQ . You are receiving this because you authored the thread.Message ID: @.***>

Issue H-3: Insufficient `onFlashloan` sanitization, combined with `obRouter` untrusted arbitrary `executor` allows performing operations on behalf of other users

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/163

Summary

`onFlashLoan` simply verifies that the `nonReentrant` modifier in `automaticLoopingAddCollateral` was enabled

```
function onFlashLoan(
    address, /* initiator */
    address collVault,
    uint amount,
    uint fee,
    bytes calldata data
) external returns (bytes32) {
    // Only callable if LeverageRouter initiates the flash loan
   /// @audit need to verify initiator and msg.sender
    assembly {
        if iszero(tload(0)) { revert(0, 0) } /// @audit this doesn't guarantee this
 ↪  came off of FL | This should be consumed on call
    }
```

This doesn't prevent multiple calls to `onFlashloan` with forged data

Meaning that once we find a way to regain control as an attacker, we can exploit the system by borrowing from other accounts, the `obRouter` allows an attacker that exact opportunity, because it allows us to pass any arbitrary `executor` address as part of `swap`

Vulnerability Detail

- obRouter allows arbitrary executor, which allows reentrancy
- The reentrancy guard is not sufficient to ensure that the flashloan came from this contract
- There's a lack of validation to prove that the call came from a `denManager` and that the flashloan callback is a callback from a call we initated

We can perform a basic Flashloan with small amounts, and then reenter via a malicious executor

From there, we can spam call `onFlashLoan` with malicious data that will trigger the `_increaseCollateral` branch

This will mint the debt to this contract

Meaning we just have to fulfill the rest of the requirements, and we will have stolen borrows from other accounts

Need to polish but seems to be a mix of issues due to `obRouter` as well as the lax flashloan calllback check

Impact

Performing borrows paid by other users

Code Snippet

POC

- Start Flashloan to open a new Den
- Pass a malicious `executor` as part of the calldata to `obRouter.swap`
- The `executor` gains control
- The `rentrancyLock` in `onFlashloan` is bypassed, since we can perform more than one call to it
- We pass `onFlashloan(data of other users)`
- We now borrow on their behalf
- We funnel the funds by either sweeping what we can or sandwiching the rest (by passing lax slippage checks)

We stole funds that belonged to other people

Tool used

Manual Review

Recommendation

The `onFlashloan` checks must be made tighter to ensure that this callback is exclusively being fulfilled as part of the intended flow

My advice:

- Store the current `DenManager` being called, ensure the callback is performed by it (msg.sender == currentDen && _requireCallerIsDenManager(msg.sender)

- Reset the reentrancy lock after the first call to `onFlashloan` (tstore(0) (or use a second transient variable)

This will most likely need to be re-review as it's a very delicate change

Discussion

dmitriia

PR61 looks ok to me

Issue H-4: BrimeDen will be instantly liquidated when opening a Den

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/170

Summary

BrimeDen is a contract that has some special perks within the protocol. One of those perks is that BrimeDen can have a Den with an ICR lower than the MCR. However, the liquidation logic has not been adapted to BrimeDen and this will cause an instant liquidation when BrimeDen is opened.

Vulnerability Detail

The objective of BrimeDen is to inject liquidity into the protocol in a cheap way, and that is the reason why BrimeDen can have an ICR lower than MCR so that the collateral requirement is not that high.

However, the `LiquidationManager` contract has not been adapted to BrimeDen so it treats it as a normal borrower. This will cause anyone to have the power to liquidate BrimeDen as soon as it opens a Den, pocketing a substantial reward for that liquidation.

Impact

As soon as BrimeDen opens a Den with an ICR lower than the MCR, anyone can liquidate it for a profit.

This will cause a direct loss of funds for the protocol as they will lose 0.5% of BrimeDen's collateral to pay out the liquidation rewards. This percentage is low but it may be a high amount given that BrimeDen is designed to inject substantial liquidity into a `DenManager`.

Code Snippet

```
function liquidateDens(IDenManager denManager, uint256 maxDensToLiquidate, uint256
↪  maxICR) public {
   // ...

    while (densRemaining > 0 && denCount > 1) {
        // ...
>>      } else if (ICR < denManagerValues.MCR) {
            singleLiquidation = _liquidateNormalMode(
                denManager,
                account,
                debtInStabPool,
```

```
                denManagerValues.sunsetting
            );
            // ...
    }
}

function batchLiquidateDens(IDenManager denManager, address[] memory _denArray)
 ↪  public {
    // ...
    while (denIter < length && denCount > 1) {
        // ...
>>      } else if (ICR < denManagerValues.MCR) {
            singleLiquidation = _liquidateNormalMode(
                denManager,
                account,
                debtInStabPool,
                denManagerValues.sunsetting
            );
            // ...
    }

    if (denIter < length && denCount > 1) {
        // ...
>>          } else if (ICR < denManagerValues.MCR) {
                singleLiquidation = _liquidateNormalMode(
                    denManager,
                    account,
                    debtInStabPool,
                    denManagerValues.sunsetting
                );
            // ...
    }
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to adjust the liquidation functions so they take into account if the Den to be liquidated is the BrimeDen and apply a lower MCR to it.

Issue H-5: `LSPRouter` Allows Attackers to Steal Approvals from Other Users

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/183

Summary

Several functions in `LSPRouter` allow attackers to steal funds from other users by exploiting approvals granted to the router.

Vulnerability Detail

The following functions are vulnerable:

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

These functions are designed to facilitate withdrawals or redemptions from the Liquid Stability Pool (LSP) on behalf of users. Before invoking these functions, a user must approve the router contract to operate on their behalf within the LSP.

The issue arises because an attacker can call any of these functions and specify another user's address as the owner parameter. If the victim currently has a non-zero approval to the router, the attacker can use this mechanism to steal all the approved funds.

Impact

Any malicious actor can exploit this vulnerability to steal funds approved to the `LSPRouter` by other users.

Code Snippet

```
function redeemPreferredUnderlying(
        ILSPRouter.RedeemPreferredUnderlyingParams calldata params
    ) external returns (uint assets, address[] memory tokens, uint[] memory
 ↪  amounts) {
        // ...
>>      assets = lsp.redeem(params.shares, params.preferredUnderlyingTokens,
 ↪  arr.receiver, params._owner);
        // ...
```

```
    }

    function redeemPreferredUnderlyingToOne(
        ILSPRouter.RedeemPreferredUnderlyingToOneParams calldata params
    ) external returns (uint assets, uint totalAmountOut) {
        // ...
>>      assets = lsp.redeem(params.shares, params.preferredUnderlyingTokens,
↪   arr.receiver, params._owner);
        // ...
    }

    function redeem(
        ILSPRouter.RedeemWithoutPrederredUnderlyingParams calldata params
    ) external returns (uint assets, address[] memory tokens, uint[] memory
↪   amounts) {
        // ...
>>      assets = lsp.redeem(params.shares, arr.receiver, params._owner);
        // ...
    }

    function withdraw(
        ILSPRouter.WithdrawFromlspParams calldata params
    ) external returns (uint shares, address[] memory tokens, uint[] memory
↪   amounts) {
        // ...
>>      shares = lsp.withdraw(params.assets, arr.receiver, params._owner);
        // ...
    }
```

Tool used

Manual Review

Recommendation

To mitigate this issue, the router should only allow users to perform actions on their own behalf. Specifically, replace the `params._owner` parameter with `msg.sender` to ensure the caller is the sole actor in these operations.

Issue H-6: The total active debt is not correctly updated when a Den is opened

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/190

Summary

The total active debt in a `DenManager` is not correctly updated when a new Den is opened, resulting in the complete removal of the interest accrued since the last update.

Vulnerability Detail

When a Den is opened within a `DenManager`, the total active debt is initially updated through `_accrueActiveInterests`. This function calculates and adds the accumulated interest since the last update to the total active debt. However, this update is subsequently ignored and overwritten when the total active debt is recalculated as the sum of the previous debt and the new debt from the opening Den.

This final update to the total active debt completely overwrites the earlier interest accrual, effectively removing it from the total active debt calculation.

Impact

This issue causes the `totalActiveDebt` variable to become desynchronized from the actual total active debt in the `DenManager`. While the individual Den debt balances remain accurate (as `activeInterestIndex` remains unaffected), the sum of these balances will exceed the value of `totalActiveDebt`.

This discrepancy can lead to the following issues:

- Checks for `maxSystemDebt` will be inaccurate because they rely on the incorrect `totalActiveDebt`.
- The reward distribution system will distribute excessive rewards to users, resulting in insufficient funds for later claimants.
- The last Dens to be closed or redeemed from the `DenManager` will fail due to an underflow in `totalActiveDebt`.

Code Snippet

```
uint256 _newTotalDebt = supply + _compositeDebt;
require(_newTotalDebt + defaultedDebt <= maxSystemDebt, "Collateral debt limit
 ↪  reached");
```

```
totalActiveDebt = _newTotalDebt;
```

PoC

The following test can be added to `DenManager.t.sol` to demonstrate that `totalActiveDebt` does not account for the interest accrued when opening a new Den:

```
function test_interest_removed_opening_den() public {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");

    _openDen(user1);

    // After a day, some interest has accrued
    vm.warp(block.timestamp + 1 days);

    // 1.005 is the initial debt and the rest is interest
    uint256 totalDebtBefore = IDenManager(wBERADenManager).getEntireSystemDebt();
    assertEq(totalDebtBefore, 1.005027534246575342e18);

    _openDen(user2);

    // After other user opens a Den, the interest accrued is removed from
↪   totalActiveDebt
    uint256 totalDebtAfter = IDenManager(wBERADenManager).getEntireSystemDebt();
    // The total debt now is only the initial debt from both Dens (1.005 + 1.005).
↪   The interest accrued has been deleted
    assertEq(totalDebtAfter, 2.01e18);
}
```

Tool used

Manual Review

Recommendation

To resolve this issue, ensure `totalActiveDebt` is correctly updated when a Den is opened:

```
-       uint256 _newTotalDebt = supply + _compositeDebt;
+       uint256 _newTotalDebt = totalActiveDebt + _compositeDebt;
        require(_newTotalDebt + defaultedDebt <= maxSystemDebt, "Collateral debt
↪   limit reached");
        totalActiveDebt = _newTotalDebt;
```

Issue H-7: StableBexFeed uses post-expected-arbitrage pricing on the LP possibly leading to massive loss on redemption

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/226

Summary

After running an invariant testing suite against the oracle, we were able to demonstrate that when the pool is imbalanced, the value received by a redeemer far exceeds the value that the StableBexFeed is outputting

Vulnerability Detail

The Formula used by `StableBexFeed` provides a floor, however, to protect redemptions against value leaks, the formula needs to provide the ceiling of the BPT Value

In lack of that, the current formula is underpricing the BPT anytime the underlying balances are not equal, meaning that the BPT has some arbitrage that hasn't been closed

Impact

The impact was demonstrated by running echidna, against a newly deployed Balancer MetaStablePool

The run was run against mainnet using Recon

The results from the run are here:
https://staging.getrecon.xyz/shares/38204c44-bf2b-4b8a-a607-bbc14b48317b

Code Snippet

```
//forge test --match-test test_optimize_max_value_underestimated_1e18_1 --fork-url
↪  https://eth-mainnet.g.alchemy.com/v2/ST0ZewmedZBEsMSxL96YZAhkpCcXOLCC -vvvv
function test_optimize_max_value_underestimated_1e18_1() public {

   set_max_value_underestimated_1e18();

   set_max_value_underestimated_1e18();

   set_max_value_underestimated_1e18();
```

```
    set_max_value_underestimated_1e18();

    set_max_value_underestimated_1e18();

    balancer_supply(727248,4771610876816214667904070494920506);

    set_max_value_underestimated_1e18();

    /**
          [559] tERC20::balanceOf(CryticToFoundry:
 ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
          ← [Return] 9999999999999998002500000000759007849 [9.999e35]
      [559] tERC20::balanceOf(CryticToFoundry:
 ↪  [0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496]) [staticcall]
          ← [Return] 995228389123183602690434872927982455 [9.952e35]
      [0] console::log("sumOfValue", 1814858338944182646138 [1.814e21]) [staticcall]
          ← [Stop]
      [0] console::log("fromOracle", 1000133367690920573 [1e18]) [staticcall]
          ← [Stop]
      ← [Stop]
    */

}
```

Tool used

Manual Review

Recommendation

Extra info

See the TargetFunctions used

```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {BaseTargetFunctions} from "@chimera/BaseTargetFunctions.sol";
import {BeforeAfter} from "./BeforeAfter.sol";
import {Properties} from "./Properties.sol";
import {vm} from "@chimera/Hevm.sol";
import {IPool} from "src/IPool.sol";

abstract contract TargetFunctions is
    BaseTargetFunctions,
    Properties
{
```

```
// The real test is to compare the Price from the BeraOracle
// Vs the Price form th
// fetchPrice to return the value of the BPT (expressed as underlying)
// withdraw of the same amount of BPT to determine the value as a proportion of
↪ it

// Handlers to LP
// Handlers to Withdraw
// Hanlders to Swap (even incorrectly)

// VERY GENERIC HANDLERS
// TODO: Change to be more clamped

/// === UNCLAMPED HANDLERS === ///
function pool_batchSwap(uint8 kind, IPool.BatchSwapStep[] memory swaps,
↪ address[] memory assets, IPool.FundManagement memory funds, int256[] memory
↪ limits, uint256 deadline) public {
    pool.batchSwap(IPool.SwapKind(kind), swaps, assets, funds, limits,
↪ deadline);
}

function pool_exitPool(address sender, address recipient, IPool.ExitPoolRequest
↪ memory request) public {
    pool.exitPool(poolId, sender, recipient, request);
}

function pool_joinPool(address sender, address recipient, IPool.JoinPoolRequest
↪ memory request) public {
    pool.joinPool(poolId, sender, recipient, request);
}

function pool_mint(address to) public {
    pool.mint(to);
}

/// === CLAMPED HANDLERS === ///
function balancer_Swap(uint256 amountIn, bool zerForOne)
    internal
    returns (uint256)
{
    IPool.BatchSwapStep[] memory steps = new IPool.BatchSwapStep[](1);
    steps[0] = IPool.BatchSwapStep(
        poolId,
        0,
        1,
        amountIn,
        abi.encode("") // Empty user data
    );
```

```
        address[] memory tokens = new address[](2);
        tokens[0] = zerForOne ? address(weth) : address(reth);
        tokens[1] = zerForOne ? address(reth) : address(weth);

        int256[] memory limits = new int256[](2);
        limits[0] = type(int256).max;
        limits[1] = type(int256).max;

        int256[] memory res = vault.batchSwap(
            IPool.SwapKind.GIVEN_IN, steps, tokens, IPool.FundManagement(owner,
 ↪  false, payable(owner), false), limits, block.timestamp
        );


        // Negative means we receive those tokens
        if (res[1] > 0) {
            revert("invalid result");
        }

        uint256 amtOut = uint256(-res[1]);

        return amtOut;
    }

    // @dev Allows a random single sided supply into the vault
    // Does not care about slippage
    function balancer_supply(uint256 amount0In, uint256 amount1In) public {
        uint256[] memory amountsIn = new uint256[](2);
        amountsIn[0] = amount0In;
        amountsIn[1] = amount1In;

        vault.joinPool(
          poolId,
          address(this),
          address(this),
          IPool.JoinPoolRequest({
              assets: _poolAssets(),
              maxAmountsIn: amountsIn,
              userData: abi.encode(
                  IPool.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT, amountsIn, 0
              ),
              fromInternalBalance: false
          })
      );
    }
}
```

And Properties

```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {Asserts} from "@chimera/Asserts.sol";
import {BeforeAfter} from "./BeforeAfter.sol";
import {IPool} from "src/IPool.sol";
import "forge-std/console2.sol";
abstract contract Properties is BeforeAfter, Asserts {

    // bool public optimize_max_value_underestimated_1e18 = true;
    // bool public optimize_max_value_overestimate_1e18 = true;
    int256 public optimize_max_value_underestimated_1e18;
    int256 public optimize_max_value_overestimate_1e18;

    function set_max_value_underestimated_1e18() public returns (int256) {

        // Estimate the price given 1e18 of asset
        uint256 fromOracle = feed.fetchPrice(address(pool)); // Assume it's already
↪   the price for 1e18

        uint256[] memory minAmountsOut = new uint256[](2);

        // Do a withdrawal and see
        uint256[] memory balancesB4 = new uint256[](2);


        balancesB4[0] = weth.balanceOf(address(owner));
        balancesB4[1] = reth.balanceOf(address(owner));


        vault.exitPool(
            poolId,
            owner,
            owner,
            IPool.ExitPoolRequest(
                _poolAssets(),
                minAmountsOut,
                abi.encode(IPool.ExitKind.EXACT_BPT_IN_FOR_TOKENS_OUT, 1e18),
                false
            )
        );

        uint256[] memory deltaBalsAfter = new uint256[](2);

        deltaBalsAfter[0] = weth.balanceOf(address(owner)) - balancesB4[0];
        deltaBalsAfter[1] = reth.balanceOf(address(owner)) - balancesB4[1];

        // Evaluate by summing up and comparing to oracle
```

```
        uint256 sumOfValue = deltaBalsAfter[0] + deltaBalsAfter[1];
        console2.log("sumOfValue", sumOfValue);
        console2.log("fromOracle", fromOracle);

        if(sumOfValue > fromOracle) {
            uint256 diff = sumOfValue - fromOracle;
            if(diff > uint256(type(int256).max)) {
                // optimize_max_value_underestimated_1e18 = false;
                optimize_max_value_underestimated_1e18 = type(int256).max;
                return type(int256).max;
            }

            optimize_max_value_underestimated_1e18 = int256(diff);
            // if(diff > 454758818989369l1) {
            //     optimize_max_value_underestimated_1e18 = false;
            // }

            return int256(diff);
        }

        if(fromOracle > sumOfValue) {
            uint256 diff = fromOracle - sumOfValue;
            if(diff > uint256(type(int256).max)) {
                // optimize_max_value_overestimate_1e18 = false;
                optimize_max_value_overestimate_1e18 = type(int256).max;
                return type(int256).max;
            }

            optimize_max_value_overestimate_1e18 = int256(diff);
            // if(diff > 3994) {
            //     optimize_max_value_overestimate_1e18 = false;
            // }

            return int256(diff);
        }

        return 0;
    }
}
```

Issue H-8: Price Feed for bHONEY Allows for Arbitrage Due to Predictability and Manipulation

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/228

Summary

The price feed for bHONEY is susceptible to arbitrage because it can be predicted and manipulated based on current trades occurring within the Berps protocol.

Vulnerability Detail

The bHONEY token serves as a vaulted version of HONEY, the official Berachain stablecoin. It is primarily used as liquidity within Berps, a perpetuals protocol, in exchange for trading fees. The bHONEY <> HONEY exchange rate is determined by the amount of HONEY tokens in the vault, which fluctuates based on the profitability of trades executed on Berps.

- **When a trade closes with profits**: The sendAssets function is called to transfer HONEY from the vault to the trader, reducing collateralization and decreasing the exchange rate.
- **When a trade closes with losses**: The receiveAssets function transfers HONEY to the vault, increasing collateralization and the exchange rate.

This exchange rate is directly used by Beraborrow to determine the price of bHONEY in USD through the bHONEYFeed contract:

```
function fetchPrice() external view returns (uint) {
    /// @dev bHONEY price is calculated on spot by now, could possibly change when
 ↪  we have a price feed
    return bHoney.shareToAssetsPrice() * priceFeed.fetchPrice(honey) / WAD;
}
```

The core issue lies in the fact that the shareToAssetsPrice value is directly influenced by trades on Berps. This allows attackers to manipulate the rate and exploit arbitrage opportunities in Beraborrow. Below are three scenarios illustrating how the bHONEY price can be artificially increased, decreased, or manipulated through swing trading.

Scenario 1: bHONEY Price Rises

1. Bob opens a large trade on Berps with unrealized losses.

2. He mints or flash-loans NECT and deposits them into the LSP (which holds `bHONEY` from previous liquidations).
3. When Bob's position is closed, the unrealized losses are realized, increasing the `bHONEY` rate.
4. Bob withdraws from the LSP, profiting from the now higher `bHONEY` price.

Scenario 2: `bHONEY` Price Falls

1. Alice opens a large trade on Berps with unrealized profits.
2. She deposits `bHONEY` as collateral in a Den on Beraborrow and borrows the maximum amount of debt.
3. When Alice's position closes with profits, the `bHONEY` rate decreases.
4. The resulting drop in `bHONEY` price causes bad debt in the Den, allowing Alice to self-liquidate and profit while distributing the bad debt across other Dens.

Scenario 3: Swing Trading

1. Charlie opens two large positions in opposite directions (long and short) on Berps.
2. Over time, one position accrues unrealized profits while the other accrues unrealized losses.
3. Charlie closes the profitable trade, reducing the `bHONEY` price.
4. He deposits into the LSP while the `bHONEY` price is deflated.
5. Charlie then closes the losing trade, increasing the `bHONEY` price.
6. Finally, Charlie withdraws from the LSP, profiting from the price recovery.

The profitability of these attacks depends on whether the arbitrage gains outweigh the fees paid to Berps and Beraborrow.

Currently, the `bHONEY` price is not updated atomically when trades close but instead on the next call to the `bHONEY` vault. This delay enables attackers to exploit trades closed by other users, such as whales, further increasing the impact of the vulnerability.

Impact

The ability to predict and manipulate the `bHONEY` price allows attackers to perform arbitrage on Beraborrow. By exploiting this mechanism, attackers can drain funds from legitimate users who have deposits in the LSP or open Dens.

PoC

The following test demonstrates how trades closed on Berps directly influence the `bHONEY` rate. Additionally, it highlights that the rate does not update immediately, creating an opportunity for arbitrage.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import {Test, console} from "forge-std/Test.sol";
interface IbHONEY {
    function shareToAssetsPrice() external view returns (uint256);
    function receiveAssets(uint256 amount, address from) external;
    function distributeReward(uint256 amount) external;
}
interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
}

contract bHONEYTest is Test {

    IbHONEY public bHONEY = IbHONEY(0x1306D3c36eC7E38dd2c128fBe3097C2C2449af64);
    IERC20 public HONEY = IERC20(0x0E4aaF1351de4c0264C5c7056Ef3777b41BD8e03);

    function setUp() public {
        // string memory RPC_URL = "https://bartio.rpc.berachain.com/";
        string memory RPC_URL = "https://bera-testnet.nodeinfra.com";
        vm.createSelectFork(RPC_URL);
    }

    function test_arb() public {
        uint256 rateBefore = bHONEY.shareToAssetsPrice();

        uint256 lossAmount = 10_000e18;
        deal(address(HONEY), address(this), lossAmount);
        HONEY.approve(address(bHONEY), lossAmount);

        // Here, we simulate a user closing a position with a loss, which is sent
 ↪  to the bHONEY vault
        bHONEY.receiveAssets(lossAmount, address(this));

        // The rate is the same even thought bHONEY is now more collateralized
 ↪  (rate should increase)
        uint256 rateAfter = bHONEY.shareToAssetsPrice();
        assertEq(rateBefore, rateAfter);

        // Here a user would make an arbitrage on Beraborrow and then call
 ↪  distributeReward to increase the rate
        bHONEY.distributeReward(0);
```

```
        uint256 rateFinal = bHONEY.shareToAssetsPrice();
        assertGt(rateFinal, rateAfter);
    }
}
```

Tool used

Manual Review

Recommendation

1. Avoid using a spot price for `bHONEY`, as it is directly influenced by trades on Berps and easily manipulable.
2. While a Chainlink price feed would still depend on Berps trades, it could mitigate risk by preventing rate manipulation within the same block.

Overall, it needs to be studied if it's even possible to use `bHONEY` as collateral to mint NECT given that its price can be predicted and manipulated by large trades on Berps.

Issue H-9: BPT can be overvalued with `WeightedBexFeed` as total supply is understated for `ComposableStablePool` and `WeightedPool` Bex pool types

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/242

Summary

Since BEX is a Balancer V2 fork `getActualSupply()` is Balancer V2 docs recommended for `ComposableStablePool` and `WeightedPool`.

Vulnerability Detail

ComposableStablePool.sol#L1082-L1097

```
/**
 * @dev Returns the effective BPT supply.
 *
 * In other pools, this would be the same as `totalSupply`, but there are two key
↪  differences here:
 *  - this pool pre-mints BPT and holds it in the Vault as a token, and as such we
↪  need to subtract the Vault's
 *    balance to get the total "circulating supply". This is called the
↪  'virtualSupply'.
 *  - the Pool owes debt to the Protocol in the form of unminted BPT, which will be
↪  minted immediately before the
 *    next join or exit. We need to take these into account since, even if they
↪  don't yet exist, they will
 *    effectively be included in any Pool operation that involves BPT.
 *
 * In the vast majority of cases, this function should be used instead of
↪  `totalSupply()`.
 */
function getActualSupply() external view returns (uint256) {
    (, uint256 virtualSupply, uint256 protocolFeeAmount, , ) =
↪  _getSupplyAndFeesData();
    return virtualSupply.add(protocolFeeAmount);
}
```

WeightedPool.sol#L325-L344

```
/**
```

```
 * @notice Returns the effective BPT supply.
 *
 * @dev This would be the same as `totalSupply` however the Pool owes debt to the
↪  Protocol in the form of unminted
 * BPT, which will be minted immediately before the next join or exit. We need to
↪  take these into account since,
 * even if they don't yet exist, they will effectively be included in any Pool
↪  operation that involves BPT.
 *
 * In the vast majority of cases, this function should be used instead of
↪  `totalSupply()`.
 */
function getActualSupply() public view returns (uint256) {
    uint256 supply = totalSupply();

    (uint256 protocolFeesToBeMinted, ) = _getPreJoinExitProtocolFees(
        getInvariant(),
        _getNormalizedWeights(),
        supply
    );

    return supply.add(protocolFeesToBeMinted);
}
```

`totalSupply()` is now used in `WeightedBexFeed`:

WeightedBexFeed.sol#L80-L82

```
uint totalSupply = pool.totalSupply();

return invariant * mult / totalSupply;
```

Impact

As `totalSupply()` underestimates real total supply by fees not yet minted the BPT can be overpriced in the current logic, which can be used for arbitraging the protocol by atomically creating bad debt.

Recommendation

Consider updating the interface and using `getActualSupply()`:

IBalancerV2Pool.sol#L12

```
function getActualSupply() external view returns (uint256);
```

WeightedBexFeed.sol#L80-L82

```
-       uint totalSupply = pool.totalSupply();

-       return invariant * mult / totalSupply;

+       uint totalSupply = pool.getActualSupply();

+       return totalSupply > 0 ? invariant * mult / totalSupply : 0;
```

Discussion

dmitriia

Ok

Issue M-1: `InfraredCollateralVault` Delta Balance on `InfraredVault.getReward` is griefable

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/159

Summary

The function `_harvestRewards` in `InfraredCollateralVault` follows a common pattern that checks for the balances before and after calling `iVault.getReward();`

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/core/vaults/InfraredCollateralVault.sol#L120-L144

The pattern is griefable by claiming the rewards on behalf of the vault

Vulnerability Detail

From reading the documentation: https://infrared.finance/docs/developers/smart-contract-apis/infrared-vault

The function `getRewardForUser` [https://infrared.finance/docs/developers/smart-contract-apis/infrared-vault#getrewardforuser] allows anyone to claim the rewards on behalf of someone else

I will need the implementation for the vault, or I will do a fork test / test on bartio to prove the impact

https://infrared.finance/docs/developers/ibgt-staking-pool#claiming-rewards

Impact

Loss of rewards

Code Snippet

Provided by @alex-beraborrow

```
[101696] 0x1B602728805Ca854e0DFDbbBA9060345fB26bc20::getRewardForUser(0x3A1bFc78717
↪  66F5Ec089C54Bb117CDd0c5F13710)
    [15113] 0x46eFC86F0D7455F135CC9df501673739d513E982::transfer(0x3A1bFc7871766F5E
  ↪  c089C54Bb117CDd0c5F13710, 26622261692724845824 [2.662e19])
        emit Transfer(from: 0x1B602728805Ca854e0DFDbbBA9060345fB26bc20, to:
  ↪  0x3A1bFc7871766F5Ec089C54Bb117CDd0c5F13710, value: 26622261692724845824
  ↪  [2.662e19])
```

```
        ← [Return] true
    emit RewardPaid(param0: 0x3A1bFc7871766F5Ec089C54Bb117CDd0c5F13710, param1:
 ↪  0x46eFC86F0D7455F135CC9df501673739d513E982, param2: 26622261692724845824
 ↪  [2.662e19])
  ← [Return]
```

See one of my findings:
https://github.com/code-423n4/2023-07-tapioca-findings/issues/1429

Tool used

Manual Review

Recommendation

It seems like `receiveDonations` will allow handling this edge case, meaning you may simply accept this as a valid risk and be forced to manually call `receiveDonations` when this happens

Discussion

alex-beraborrow

It does indeed allow it:

```
[101696] 0x1B602728805Ca854e0DFDbbBA9060345fB26bc20::getRewardForUser(0x3A1bFc78717⌋
↪  66F5Ec089C54Bb117CDd0c5F13710)
    [15113] 0x46eFC86F0D7455F135CC9df501673739d513E982::transfer(0x3A1bFc7871766F5E⌋
 ↪  c089C54Bb117CDd0c5F13710, 26622261692724845824 [2.662e19])
        emit Transfer(from: 0x1B602728805Ca854e0DFDbbBA9060345fB26bc20, to:
 ↪  0x3A1bFc7871766F5Ec089C54Bb117CDd0c5F13710, value: 26622261692724845824
 ↪  [2.662e19])
        ← [Return] true
    emit RewardPaid(param0: 0x3A1bFc7871766F5Ec089C54Bb117CDd0c5F13710, param1:
 ↪  0x46eFC86F0D7455F135CC9df501673739d513E982, param2: 26622261692724845824
 ↪  [2.662e19])
  ← [Return]
```

Will discuss this internally if we have to adapt, or if we can have Infrared adapt the function.

dmitriia

It seems that staking needs to be added to the solution. Griefing here not only pass through the balance difference, but also the `ibgtVault` staking done in `\_autoCompoundHook()`.

So result tokens appearing on the balance due to attacker's claiming will not be in `rewardedTokens`:

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

        return (_rewards, _token);
    }
```

`_harvestRewards()`:

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

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

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

```
    function internalizeDonations(address[] memory tokens, uint128[] memory
↪   amounts) external virtual onlyOwner {
        uint tokensLength = tokens.length;

        require(tokensLength == amounts.length, "CollVault: tokens and amounts
↪   length mismatch");
```

```
        for (uint i; i < tokensLength; i++) {
            address token = tokens[i];
            uint128 amount = amounts[i];

            if (amount == 0) continue;

            uint donatedAmount = IERC20(token).balanceOf(address(this)) -
    ↪   getBalanceOfWithFutureEmissions(token);
            require(donatedAmount >= amount, "CollVault: insufficient balance");
>>          require(\_isRewardedToken(token), "CollVault: token not rewarded");  //
    ↪   @audit: fails as `ibgtVault` is in the list, not the reward token harvested

            \_getInfraredCollVaultStorage().balanceData.addEmissions(token, amount);
        }
    }
```

alex-beraborrow

Well seen @dmitriia .

How would you attack this one?

```
function internalizeDonations(address[] memory tokens, uint128[] memory amounts)
    ↪   external virtual onlyOwner {
    uint tokensLength = tokens.length;

    require(tokensLength == amounts.length, "CollVault: tokens and amounts length
    ↪   mismatch");

    IInfraredVault iVault = infraredVault();
    address \_ibgt = ibgt();
    IIBGTVault \_ibgtVault = IIBGTVault(ibgtVault());

    for (uint i; i < tokensLength; i++) {
        address token = tokens[i];
        uint128 amount = amounts[i];

        if (amount == 0) continue;

        (amount, token) = \_autoCompoundHook(token, \_ibgt, \_ibgtVault, amount);

        uint donatedAmount = IERC20(token).balanceOf(address(this)) -
    ↪   getBalanceOfWithFutureEmissions(token);
        require(donatedAmount >= amount, "CollVault: insufficient balance");
        require(\_isRewardedToken(token), "CollVault: token not rewarded");

        if (token == asset()) {
            iVault.stake(amount);
        }
```

```
        \_getInfraredCollVaultStorage().balanceData.addEmissions(token, amount);
    }
}
```

alex-beraborrow

https://github.com/Beraborrowofficial/blockend/commit/a39d79b95d7265a8f422f5d263b4d79cb694af5c

dmitriia

Looks good. It seems like `\_autoCompoundHook()` should go after the `donatedAmount >= amount` check to keep errors clear.

Also, it should fully resemble the `\_harvestRewards()` flow: now it's the way to surpass `\_performanceFee`, so there is an incentive for stakers, unrelated to griefing, to run `getRewardForUser` rewards claiming. So I think at least L144, 145 and L159-161 should be also added to `internalizeDonations()`.

alex-beraborrow

Agreed. https://github.com/Beraborrowofficial/blockend/commit/1e94298c02fab8c5b2d7365c1bbf6020f45c928a

dmitriia

Looks ok

GalloDaSballo

It seems like I disagree with @dmitriia assessment here

The `totalAssets` carry the rewards Rewards are permissionelessly claimed The `totalAssets` have decreased

An attack can gain from this

See: https://github.com/sherlock-audit/2024-11-beraborrow/issues/230

Please lmk if you see it otherwise

santipu03

Marking this issue as ”Won't Fix” for the final report

alex-beraborrow

I agree. The applied fix solves the 'Loss of rewards' impact, but agree that we'll have to remove `earned()` usage to prevent a sudden price fall. It could still cause a sudden price increase if rewards aren't griefed through `getRewardForUser`, but that's way better than the price decrease. @santipu03 @GalloDaSballo

alex-beraborrow

And this issue was indeed mitigated, the one that wasn't is #230, which my latest comment should fix.

Issue M-2: Loss of rewards in DenManager due to precision loss in `_updateMintVolume`

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/166

The protocol has acknowledged this issue.

Summary

Users that mint debt through `DenManager` will get an unfair amount of mint rewards due to precision loss happening at `_updateMintVolume`.

Vulnerability Detail

Users can mint some debt in DenManager through the functions `openDen` and `updateDenFromAdjustment`. These two functions are calling `_updateMintVolume` to accrue some rewards to the user who is taking the debt in NECT.

```
    function _updateMintVolume(address account, uint256 initialAmount) internal {
>>      uint32 amount = uint32(initialAmount / VOLUME_MULTIPLIER);
        (uint256 week, uint256 day) = getWeekAndDay();
        totalMints[week][day] += amount;

        // ...
    }
```

The variable `initialAmount` is the amount of debt that is minted by the borrower, and it gets divided by `VOLUME_MULTIPLIER`, which is a constant with the value of `1e20`:

```
// volume-based amounts are divided by this value to allow storing as uint32
uint256 constant VOLUME_MULTIPLIER = 1e20;
```

This constant is used to downscale the debt amounts so they can be stored in a `uint32` variable. However, this will cause a huge precision loss that will lead to some users not receiving any mint rewards when they mint an amount of debt lower than 100 NECT.

Imagine the following scenario:

- Bob opens a Den to mint 200 NECT
- Alice opens a Den to mint 90 NECT
- A day later, Alice adjusts the Den to get 90 NECT more
- A few days later, Alice adjusts the Den again to mint 90 NECT of extra debt
- After the week is over and mint rewards have to be distributed, Bob will receive all the mint rewards while Alice will receive none. This is unfair because Alice has

minted more debt overall but it hasn't been saved in the rewards mechanism due to this precision loss.

Impact

Users that mint an amount of debt lower than 100 NECT won't receive any mint rewards.

Also, users who mint different amounts of debt will receive the same amounts of rewards as if they got the same debt. For example, Alice and Bob would have received the same rewards if they minted 100 and 199 NECT.

Code Snippet

```
    function _updateMintVolume(address account, uint256 initialAmount) internal {
>>      uint32 amount = uint32(initialAmount / VOLUME_MULTIPLIER);
        // ...
    }
```

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to change the data type of `totalMints` and `VolumeData.amount` from `uint32` to `uint256` to allow storing higher values. This would allow us to remove entirely the variable `VOLUME_MULTIPLIER`, and thus remove the precision loss as a whole.

Discussion

alex-beraborrow

Acknowledged, we'll have a minNetDebt > 100 NECT.

santipu03

Got it.

Still, you should know that the rounding will still be present and will cause some unfair scenarios:

- Users who mint 100 NECT and users who mint 199 NECT will receive the same rewards.

- Users who mints 200 NECT will receive double the rewards than a user who mints 199 NECT.

If you still decide to acknowledge the issue regardless of the rounding, users should know about it.

alex-beraborrow

Update: We have decided to shutdown POLLEN rewards through DenManagers since of the operational burden needed with `dao/` contracts.

Issue M-3: Liquidated Dens with ICR slightly above 100% will cause a loss of funds to the `LiquidStabilityPool`

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/171

Summary

When a Den is liquidated with an ICR slightly above 100% but still below 100.5%, it will cause a loss of funds for the users in `LiquidStabilityPool` due to the collateral gas compensation.

Vulnerability Detail

A Den can be liquidated in two ways:

1. If a Den has an ICR lower than MCR but still higher than 100%, it should be liquidated using the `LiquidStabilityPool`.
2. If the Den has an ICR lower than 100% it should be liquidated without the `LiquidStabilityPool`, instead the debt and collateral are distributed within the same `DenManager`.

This distinction is made so that liquidations that carry bad debt, i.e. debt without backing collateral, do not negatively impact the `LiquidStabilityPool` and only that `DenManager`. However, there is an edge case where a Den has an ICR higher than 100% but it still generates some bad debt due to the gas compensation.

The collateral gas compensation is a percentage of the total collateral of a Den (**0.5%**) that is discounted from a Den's collateral on liquidation and sent to the liquidator (and other parties). This gas compensation can cause a Den that is slightly healthy, meaning it has an ICR slightly above 100%, to be suddenly unhealthy without that 0.5% of collateral.

Impact

This issue will happen whenever a Den is liquidated with an ICR that is above 100% but is below 100.5%. In these scenarios, the liquidation will offset the Den's debt and collateral using the `LiquidStabilityPool`, causing losses to its depositors.

If the Den is big enough, it can cause a significant loss to users in `LiquidStabilityPool`, leading to some withdrawals from there and hurting the overall protocol health.

Code Snippet

```
function liquidateDens(IDenManager denManager, uint256 maxDensToLiquidate, uint256
↪ maxICR) public {
    // ...

    while (densRemaining > 0 && denCount > 1) {
        // ...
>>      if (ICR <= _100pct) {
            singleLiquidation = _liquidateWithoutSP(denManager, account);
            _applyLiquidationValuesToTotals(totals, singleLiquidation);
        } else if (ICR < denManagerValues.MCR) {
            // ...
}


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

        // closed / non-existent dens return an ICR of type(uint).max and are
↪ ignored
        uint ICR = denManager.getCurrentICR(account, denManagerValues.price);
>>      if (ICR <= _100pct) {
            singleLiquidation = _liquidateWithoutSP(denManager, account);
        } else if (ICR < denManagerValues.MCR) {
            // ...
    }

    if (denIter < length && denCount > 1) {
        // ...
>>          if (ICR <= _100pct) {
                singleLiquidation = _liquidateWithoutSP(denManager, account);
            } else if (ICR < denManagerValues.MCR) {
                // ...
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue, the liquidation functions should liquidate without using the `LiquidStabilityPool` when the Den's ICR is below 100.5% and not 100%.

Issue M-4: The new fee on collateral vaults will be applied to previous rewards

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/172

Summary

When the performance fee is changed on a collateral vault, the new fee will be applied to previous rewards instead of only applying to future ones. This will cause a sudden change in `totalAssets`, possibly leading to unintended consequences such as unwanted liquidations.

Vulnerability Detail

In collateral vaults (`InfraredCollateralVault.sol`), there is a performance fee that is applied to the rewards that are generated within the underlying vault from Infrared. This fee is applied on `totalAssets` and `_harvestRewards`.

When the protocol wants to change this performance fee, they call the function `setPerformanceFee`.

However, this fee doesn't harvest the previous rewards accrued until this moment, and this will cause those unharvested rewards to be accrued by applying this new fee instead of the previous one.

Impact

When the unharvested rewards are significant, this issue will cause a sudden change in the value of `totalAssets` depending on the direction of the change:

- When the performance fee is changed to a lower value, it will cause a sudden increase in `totalAssets`.
- When the performance fee is changed to a higher value, it will cause a sudden decrease in `totalAssets`. This may cause an unwanted liquidation for a borrower that has an open Den with most of the collateral from this affected Vault and an ICR at the limit with MCR.

Code Snippet

```
function setPerformanceFee(uint16 _performanceFee) external virtual onlyOwner {
    IInfraredCollateralVault.InfraredCollVaultStorage storage $ =
↪   _getInfraredCollVaultStorage();
```

```
    require(_performanceFee >= $.minPerformanceFee && _performanceFee <=
 ↪  $.maxPerformanceFee, "CollVault: performance fee out of bounds");

    $.performanceFee = _performanceFee;
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to adapt the function `setPerformanceFee` so that it calls `_harvestRewards` first before changing the performance fee.

Issue M-5: Loss of funds on `InfraredCollateralVault` when the asset has less than 18 decimals

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/174

Summary

When the asset used on an `InfraredCollateralVault` has less than 18 decimals, the function `totalAssets` will return a wrong value because it will overestimate the value of the reward tokens, assuming the main asset in the Vault has 18 decimals.

This will cause users to withdraw more assets than they should from the Vault, leaving the last depositors without funds.

Vulnerability Detail

The function `totalAssets` within `InfraredCollateralVault` does some calculations to convert the value of the reward tokens in terms of the main asset in the vault. These calculations do the following:

1. Convert the balances of reward tokens (harvested or not) to USD, scaling the result to 18 decimals.
2. Convert the USD value in terms of the vault asset, scaling the result to 18 decimals.
3. Adding the calculated amount to the asset balance in the Vault.

The final result will be wrong if the asset has less than 18 decimals because the final sum is adding two values assuming both are scaled to 18 decimals when this may not be the case.

In the scenario where the asset in the vault has less than 18 decimals, the final value will be way higher than it should be because it it not scaling the asset balance to 18 decimals before the final sum.

Impact

The value of `totalAssets` will be higher than expected, leading to users withdrawing too many assets and leaving the last depositors of the Vault without funds.

Code Snippet

```
function totalAssets() public view override virtual returns (uint amountInAsset) {
    // ...

    for (uint i; i < rewardedTokensLength; i++) {
        usdValue += _convertToValue(_rewardedTokens[i], true);
    }

    // ..
>>  amountInAsset = usdValue.mulDiv(WAD, assetPrice) + assetBalance;
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to scale the asset balance to 18 decimals before adding it to the value of the rewards.

```
-   amountInAsset = usdValue.mulDiv(WAD, assetPrice) + assetBalance;
+   amountInAsset = usdValue.mulDiv(WAD, assetPrice) + (assetBalance * 10 ** (18 -
↪   decimals));
```

Issue M-6: DoS on withdrawals from Collateral Vaults due to revert in `_withdrawExtraRewardedTokens`

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/175

Summary

Some collateral vaults receive rewards in the token `iBGT` and they stake those rewards in `iBGTVault` to accrue more yield. On a withdrawal, part of these rewards are redeemed from the `iBGTVault` and sent back to our vault. However, when the amount to redeem is near zero, it will revert when trying to withdraw a 0 amount from the Infrared underlying vault, causing a temporary DoS.

Vulnerability Detail

When a user withdraws from a collateral vault, we must send the user a pro-rata share of all the assets on that vault, including the rewards earned. The function that sends out the reward tokens during a withdrawal is `_withdrawExtraRewardedTokens`.

In the case that some of the reward tokens are `iBGT`, those will be staked in `iBGTVault` so we must redeem from that vault and send the received funds to the user that is withdrawing. However, when the redeemed amount is a value near zero, the `redeem` call will revert when `iBGTVault` tries to withdraw a 0 amount from the underlying Infrared vault.

Impact

When this happens, there will be a temporary DoS on withdrawals.

This DoS will be solved when some more time passes and more rewards accrue so `iBGTVault` can withdraw a non-zero amount from its underlying vault.

Code Snippet

```
/// @dev Rewards the rest of the rewarded tokens (not the asset) to the receiver
function _withdrawExtraRewardedTokens(
    address receiver,
    uint shares,
    uint _totalSupply
) internal virtual {
    // ...
```

```
    for (uint i; i < tokensLength; i++) {
        // ...

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

Tool used

Manual Review

Recommendation

To mitigate this issue, is recommended to introduce a conditional in the `redeem` function so that it doesn't try to withdraw a zero amount from the underlying vault.

```
function redeem(uint shares, address receiver, address _owner) public override
 ↪  harvestRewards returns (uint assets) {
    // ...

    uint bHoneyAmount = shares.mulDiv(getBalance(asset()), _totalSupply,
 ↪  Math.Rounding.Down);
+   if (bHoneyAmount != 0) {
    infraredVault().withdraw(bHoneyAmount);
    _decreaseBalance(asset(), bHoneyAmount);
+   }

    _withdraw(_msgSender(), receiver, _owner, bHoneyAmount, shares);
    _withdrawExtraRewardedTokens(receiver, shares, _totalSupply);
}
```

Issue M-7: Loss of funds on `InfraredCollateralVault` if a reward token is the same as the vault asset

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/176

Summary

When a collateral vault (`InfraredCollateralVault`) receives a reward token that is the same as the vault asset, it won't deposit it again on the underlying vault when harvested, leading to a loss of yield and DoS when the last users try to withdraw from the Vault.

Vulnerability Detail

Collateral Vaults are designed to receive funds from users and deposit those in an underlying vault from Infrared to generate yield. That underlying vault distributes some reward tokens, which will be added to the total assets of the collateral vault so users can receive them when withdrawing.

However, when a collateral vault receives a reward token that is the same as the vault asset, those rewards should be deposited in the underlying vault again so that they can generate more yield. Also, they should be deposited because when a withdrawal happens, the vault assumes all the deposit tokens are staked in the underlying vault.

In the current implementation, when a collateral vault receives a reward token that is the same as the deposit asset, it won't stake it again in the underlying vault unless the collateral vault is `iBGT`. This behavior that happens only in `iBGT` vault should be the norm in all vaults to avoid this issue.

Impact

When a collateral vault does not stake the rewards in the underlying vault when those rewards are in the same token as the deposit asset, there will be a yield loss and a loss of funds for the last users of the Vault.

Code Snippet

```
function _harvestRewards() private {
    // ...
    for (uint i; i < tokensLength; i++) {
        // ...
```

```
        if (address(_ibgtVault) == address(this) && _token == _ibgt) {
>>          iVault.stake(netRewards);
        }
    }
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue, is recommended to adapt the `_harvestRewards` function so that it stakes the rewards in the underlying vault in case there are the same as the deposit token.

```
function _harvestRewards() private {
    // ...
    for (uint i; i < tokensLength; i++) {
        // ...

-       if (address(_ibgtVault) == address(this) && _token == _ibgt) {
+       if (_token == asset()) {
            iVault.stake(netRewards);
        }
    }
}
```

Issue M-8: The entry and exit fees on `LiquidStabilityPool` are lower than intended

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/180

Summary

The entry and exit fees on `LiquidStabilityPool` are a safeguard to prevent MEV bots from profiting off the protocol users through arbitrage. However, a wrong implementation will cause these fees to be lower than intended, failing to fully protect legitimate users from MEV bots.

Vulnerability Detail

Whenever a user deposits or withdraws from `LiquidStabilityPool`, some fees are taken from the funds used and sent to the protocol. The calculation of those fees is implemented in the functions `_feeOnRaw` and `_feeOnTotal`, which are called by the functions `previewDeposit`, `previewMint`, `previewWithdraw`, and `previewRedeem`.

However, the preview functions are calling the wrong fee functions so the final entry and exit fees are wrongly calculated, leading to a lower fee than intended.

Imagine the following scenario:

- Entry fee is 10%
- User deposits 100 (`100e18`) NECT into `LiquidStabilityPool`:
  - Function `previewDeposit` will return ~90.91 shares instead of 90 shares.
- User will receive 90.9 shares, which are equivalent to 90.9 assets.
  - The entry fee has been ~9.09 NECT (`100 - 90.91`)

At the end of the transaction, the user has deposited 100 NECT and has received 90.91 shares, which is equivalent to 90.91 NECT. Therefore, **the fee has been 9.09 NECT, which is equivalent to 9.09% instead of the 10%** specified by the protocol.

Impact

The entry and exit fees will be lower than intended by the protocol.

Code Snippet

```
/// @dev Calculates the fees that should be added to an amount `shares` that does
↪ already include fees.
/// Used in {IERC4626-deposit}, {IERC4626-mint}, {IERC4626-withdraw} and
↪ {IERC4626-previewRedeem} operations.
function _feeOnRaw(
    uint shares,
    uint feeBP
) private pure returns (uint) {
    return shares.mulDiv(feeBP, BP, Math.Rounding.Up);
}

/// @dev Calculates the fee part of an amount `shares` that deoes not includes fees.
/// Used in {IERC4626-previewDeposit} and {IERC4626-previewRedeem} operations.
function _feeOnTotal(
    uint shares,
    uint feeBP
) private pure returns (uint) {
    return shares.mulDiv(feeBP, feeBP + BP, Math.Rounding.Up);
}
```

PoC

The following tests demonstrates the the actual entry and exit fees are lower than intended. The tests can be pasted in the file `test/core/LiquidStabilityPool/01-deposit/deposit.t.sol`.

```
function test_fee_deposit() public {
    // Change entry fee to 10% on core
    vm.prank(owner);
    beraborrowCore.setEntryFee(0.1e4);

    // User deposits 100e18
    deal(address(nectarToken), address(depositor), 100e18);
    vm.startPrank(depositor);
    nectarToken.approve(address(liquidStabilityPool), 100e18);
    liquidStabilityPool.deposit(100e18, depositor);
    vm.stopPrank();

    assertEq(liquidStabilityPool.balanceOf(depositor), 90.909090909090909090e18);

    // assets == shares
    uint256 fee = 100e18 - liquidStabilityPool.balanceOf(depositor);

    // The expected fee is 10% but the real fee is ~9.09%
    assertEq(fee, 9.090909090909090910e18);
}
```

```
function test_fee_redeem() public {
    // Change entry fee to 0% and exit fee to 10% on core
    vm.startPrank(owner);
    beraborrowCore.setEntryFee(0);
    beraborrowCore.setExitFee(0.1e4);
    vm.stopPrank();

    // User deposits 100e18
    deal(address(nectarToken), address(depositor), 100e18);
    vm.startPrank(depositor);
    nectarToken.approve(address(liquidStabilityPool), 100e18);
    liquidStabilityPool.deposit(100e18, depositor);
    vm.stopPrank();

    assertEq(liquidStabilityPool.balanceOf(depositor), 100e18);

    // Save balance of depositor
    uint256 balanceBefore = nectarToken.balanceOf(depositor);

    // User redeems all shares
    vm.prank(depositor);
    liquidStabilityPool.redeem(100e18, depositor, depositor);

    uint256 receivedNect = nectarToken.balanceOf(depositor) - balanceBefore;
    uint256 fee = 100e18 - receivedNect;

    // The expected fee is 10% but the real fee is ~9.09%
    assertEq(fee, 9.090909090909090910e18);
}
```

Tool used

Manual Review

Recommendation

To mitigate this issue it is recommended to change the implementation of `previewDeposit`, `previewMint`, `previewWithdraw` and `previewRedeem` so they take the correct fee instead of a lower one.

Discussion

santipu03

The fix has messed even more the fee calculations on deposits and mints. Withdrawals and redeems seems to work correcty, however is recommended to make stricter tests regarding fee calculations to ensure no bugs remain.

Take a look at the following modified tests to see the issue:

```
    function test\_fee\_deposit() public {
        // Change entry fee to 10\% on core
        vm.prank(owner);
        beraborrowCore.setEntryFee(0.1e4);

        assertEq(liquidStabilityPool.totalAssets(), 0);
        assertEq(liquidStabilityPool.totalSupply(), 0);

        // User deposits 100e18
        deal(address(nectarToken), address(depositor), 100e18);
        vm.startPrank(depositor);
        nectarToken.approve(address(liquidStabilityPool), 100e18);
        liquidStabilityPool.deposit(100e18, depositor);
        vm.stopPrank();

        assertEq(liquidStabilityPool.balanceOf(depositor), 90e18);

        // assets == shares
        uint256 fee = 100e18 - liquidStabilityPool.balanceOf(depositor);

        assertEq(fee, 10e18);

        // @audit The actual fee has been 9e18 instead of 10e18
>>      assertEq(liquidStabilityPool.balanceOf(feeReceiver), 9e18);
>>      assertEq(liquidStabilityPool.totalAssets(), 100e18);
>>      assertEq(liquidStabilityPool.totalSupply(), 99e18);
    }
```

```
    function test\_fee\_mint() public {
        // Change entry fee to 10\% on core
        vm.prank(owner);
        beraborrowCore.setEntryFee(0.1e4);

        assertEq(liquidStabilityPool.totalAssets(), 0);
        assertEq(liquidStabilityPool.totalSupply(), 0);

        // User deposits 100e18
        uint assets = liquidStabilityPool.previewMint(100e18);
        deal(address(nectarToken), address(depositor), assets);
        vm.startPrank(depositor);
        nectarToken.approve(address(liquidStabilityPool), assets);
        liquidStabilityPool.mint(100e18, depositor);
        vm.stopPrank();
```

```
        assertEq(liquidStabilityPool.balanceOf(depositor), 100e18);
        assertEq(nectarToken.balanceOf(depositor), 0);

        uint256 fee = liquidStabilityPool.balanceOf(feeReceiver);

        assertEq(fee, 10e18);

        // @audit The user has transfered 111 assets but only 110 shares are minted
>>      assertEq(liquidStabilityPool.totalAssets(), 111.111111111111111112e18);
>>      assertEq(liquidStabilityPool.totalSupply(), 110e18);
    }
```

alex-beraborrow

I noticed, this PR should fix it:
https://github.com/Beraborrowofficial/blockend/pull/106/files

Tests covering you exact concerns you shared:
https://github.com/Beraborrowofficial/blockend/blob/7026974f3767e95d49a38e97e2d5c3df062b78a2/test/core/LiquidStabilityPool/01-deposit/deposit.t.sol allowbreak #L66C1-L185C6

santipu03

Looks good

Issue M-9: BrimeDen won't absorb any redemptions if its ICR is lower than the normal MCR

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/181

Summary

The BrimeDen is a special contract that is designed to have a lower MCR and not pay any interest as a way for the protocol to inject liquidity cheaply in a `DenManager` and absorb most of the redemptions. However, due to a lack of adapting the redemption functions for the BrimeDen, this won't absorb any redemptions given its ICR is lower than the usual MCR.

Vulnerability Detail

When a redemption happens within a `DenManager`, the Den with the lowest ICR is the first to get redeemed. However, if a Den's ICR is lower than the MCR, it won't get redeemed at all because the system considers it should be liquidated instead of redeemed.

This behavior does not take into account that BrimeDen is allowed to have an ICR lower than the usual MCR and not get liquidated for it. Therefore, the current implementation won't allow BrimeDen to absorb any redemption at all, defeating the whole purpose of BrimeDen.

Impact

BrimeDen won't be absorbing any redemption, which is contrary to its whole purpose for the protocol. This will cause other borrowers to get redeemed first, causing a bad experience for those given that they may prefer to keep open their Dens to earn extra rewards.

Code Snippet

```
    function redeemCollateral(
        uint256 _debtAmount,
        address _firstRedemptionHint,
        address _upperPartialRedemptionHint,
        address _lowerPartialRedemptionHint,
        uint256 _partialRedemptionHintNICR,
        uint256 _maxIterations,
        uint256 _maxFeePercentage
```

```
    ) external {
        // ...

>>      if (_isValidFirstRedemptionHint(_sortedDensCached, _firstRedemptionHint,
 ↪  totals.price, _MCR)) {
            currentBorrower = _firstRedemptionHint;
        } else {
            currentBorrower = _sortedDensCached.getLast();
>>          // Find the first den with ICR >= MCR
>>          while (currentBorrower != address(0) && getCurrentICR(currentBorrower,
 ↪  totals.price) < _MCR) {
                currentBorrower = _sortedDensCached.getPrev(currentBorrower);
            }
        }

        // ...
    }
```

```
    function _isValidFirstRedemptionHint(
        ISortedDens _sortedDens,
        address _firstRedemptionHint,
        uint256 _price,
        uint256 _MCR
    ) internal view returns (bool) {
        if (
            _firstRedemptionHint == address(0) ||
            !_sortedDens.contains(_firstRedemptionHint) ||
>>          getCurrentICR(_firstRedemptionHint, _price) < _MCR
        ) {
            return false;
        }

        address nextDen = _sortedDens.getNext(_firstRedemptionHint);
        return nextDen == address(0) || getCurrentICR(nextDen, _price) < _MCR;
    }
```

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to allow the redemption of Dens that have an ICR lower than MCR but still higher than 100%.

Instead, if the protocol prefers to keep the current redemption behavior, is recommended to introduce a conditional that allows BrimeDen to absorb a redemption even if its ICR is lower than the normal MCR.

Issue M-10: The LSP offers risk free yield to depositors that do not contribute to Liquidations

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/182

The protocol has acknowledged this issue.

Summary

LSP `offset` will start vesting after a liquidation / liquidations happened Socializing yield to non useful deposits

Vulnerability Detail

The LSP `offset` works as follows:

- It will first determine the value of the collateral being received, and the debt being paid
- And then determine what the surplus from the collateral is

This allows the LSP to vest the rewards over time

Due to this mechanism, depositors that deposit after liquidations have happened, will receive the rewards, without having contribute to the liquidations

Impact

This effectively allows new depositors to contribute close to nothing to the economic security of the protocol, while receiving yield for staking

Tool used

Manual Review

Recommendation

I'm not confident that having new depositors deposit as if all rewards were vested, as it opens up to this issue

Repricing of vested assets could cause issues and be problematic

I had suggested having new depositors deposit with assets counting all the rewards as vested

By doing that we're preventing them from gaining risk free yield

However, the downside of that is that other assets are priced differently, meaning that this could also cause losses to depositors as they are not fundamentally getting the same thing

Discussion

alex-beraborrow

I would fix it by returning back to our previous implementation, which for deposit/mint included vesting amount into totalAssets. Depositors will last until it's fully unlocked to get the price they paid when withdrawing, but it's a minor effect.

santipu03

Marking a ”Won't Fix” as the actual fix will come in a future release

Issue M-11: Impossible to liquidate borrower when a DenManager instance only has 1 active borrower

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/184

The protocol has acknowledged this issue.

Summary

It's not possible to liquidate a borrower in a `DenManager` when it's the only open Den within that instance.

Vulnerability Detail

For the record, this issue has been reported by Cyfrin in their audit of Bima Money, which is a protocol forked from Prisma Finance, therefore having a lot of similarities with Beraborrow. The link to that audit is here.

The issue is that the `LiquidationManager` doesn't allow liquidations to happen when there is only one open Den on that `DenManager`. In that scenario, the Den cannot be redeemed nor liquidated, leading to the accumulation of bad debt within the protocol.

Impact

If this issue happens while the `DenManager` is sunsetting, that Den will stay there leading to a worse overall TCR for the protocol, damaging Beraborrow as a whole.

When there is no sunsetting, the Den will be liquidated when other Dens are opened in that `DenManager`, but the late liquidation will likely create some bad debt.

Code Snippet

```
>>  function liquidateDens(IDenManager denManager, uint256 maxDensToLiquidate,
 ↪  uint256 maxICR) public {
        // ...

>>      while (densRemaining > 0 && denCount > 1) {
            // ...
        }

>>      if (densRemaining > 0 && !denManagerValues.sunsetting && denCount > 1) {
            // ...
```

```
        }
    }
```

Tool used

Manual Review

Recommendation

To mitigate this issue, the best solution would be for the protocol to always have one open Den in all active `DenManagers` so that this issue does not happen.

Even if this recommendation is applied, it would be possible that when a `DenManager` is sunsetting, the last Den to be redeemed has its ICR move below MCR, leading to having an unliquidatable and unredeemable Den.

Discussion

alex-beraborrow

We will handle this by having or own trove on each TroveManager with minNetDebt and highest CR. This ensures everyone is liquidatable and also during sunsetting we can just close our own trove which will be the final one.

Issue M-12: `InfraredCollateralVault` can be easily arbitraged due to lack of entry/exit fees

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/189

Summary

The collateral vaults on Beraborrow (named `InfraredCollateralVault`) are too prone to arbitrage due to the lack of entry and exit fees.

Vulnerability Detail

These arbitrage opportunities can be caused by different scenarios:

- The owner calling `rebalance` with some slippage
- Oracle drift

In the above cases, an MEV bot can take profit from a situation by depositing and withdrawing a ton of assets into a collateral vault in the same transaction, even incrementing the damage by leveraging a flash loan.

Impact

MEV bots can extract profit from collateral vaults causing losses to legitimate users.

Tool used

Manual Review

Recommendation

It's recommended to implement entry and exit fees on collateral vaults to prevent MEV bots from extracting profits at the expense of legitimate users.

Issue M-13: Some rewards within `DenManager` will end up locked due to a flawed design

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/192

The protocol has acknowledged this issue.

Summary

The current system in `DenManager` for distributing rewards contains a flaw that causes some rewards to become permanently stuck and unrecoverable. The root cause is that the rewards are distributed based on the debt balances without applying first the interest accrued.

Vulnerability Detail

The design of the reward distribution system operates as follows:

1. Whenever an update occurs in `DenManager`, the `_updateRewardIntegral` function is called to update the `rewardIntegral` variable, which tracks rewards owed per unit of debt.
   - The calculation for updating `rewardIntegral` is `rate * duration * 1e18 / supply`.
   - The `supply` variable represents the total active debt **before accounting for accrued interest.**

```
    function _updateRewardIntegral(uint256 supply) internal returns (uint256
 ↪  integral) {
        // ...
>>          integral += (duration * rewardRate * 1e18) / supply;
        // ...

        return integral;
    }
```

2. The `_updateIntegralForAccount` function is then called to update the rewards owed to an individual Den.
   - The pending reward for a Den is calculated as `balance * (currentIntegral - userIntegral) / 1e18`.
   - The `balance` variable represents the debt balance of a Den **before accounting for accrued interest.**

```
    function _updateIntegralForAccount(address account, uint256 balance, uint256
↪   currentIntegral) internal {
        uint256 integralFor = rewardIntegralFor[account];

        if (currentIntegral > integralFor) {
>>          storedPendingReward[account] += (balance * (currentIntegral -
↪   integralFor)) / 1e18;
            rewardIntegralFor[account] = currentIntegral;
        }
    }
```

The flaw arises because rewards are distributed based on the debt **before interest is accrued**. Since `totalActiveDebt` is updated more frequently than individual Den balances, rewards distributed via `rewardIntegral` use a total active debt value that includes accrued interest, while individual Den balances do not. This mismatch results in an inconsistency in reward distribution.

Example Scenario

1. Bob and Alice each have `100e18` in debt within the `DenManager`, making the total active debt `200e18`.
2. At the start of the week, `1e18` rewards are distributed.
3. Midweek, Bob calls `claimReward`:
    - Half of the rewards (`0.5e18`) are distributed based on the total debt without recent accrued interest (`100e18`).
    - A 5% interest is applied, increasing Bob's debt to `105e18` and the total debt to `210e18`.
    - Bob claims `0.5e18` rewards, which is correct.
4. By the end of the week, both Bob and Alice call `claimReward`:
    - The remaining rewards (`0.5e18`) are distributed based on the updated total debt (`210e18`).
    - Bob correctly claims `0.5e18`, but Alice claims less than `0.5e18` because her rewards are calculated based on her debt before the interest accrual (`100e18`), even though the integral was updated as if her debt was `105e18`.

Impact

This flaw results in some rewards being locked in the `DenManager`. The higher the interest rate, the more significant the amount of locked rewards.

Users who frequently update their Dens will receive a larger share of the rewards, while less active users will lose out on their fair share.

PoC

The following test can be run in `DenManager.t.sol` and it shows how the interest accrued will make some users lose rewards.

The test requires a mock to simulate and simplify the behavior of the Vault that distributes rewards:

```
// Contract to mock the Vault
contract VaultMock {
    function allocateNewEmissions(uint256) public pure returns (uint256) {
        return 1e18;
    }

    function transferAllocatedTokens(address , address , uint256 amount) public
↪   pure returns (bool) {
        console2.log("transfered:  %e", amount);
        return true;
    }
}
```

```
function test_locked_rewards() public {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");

    _openDen(user1);
    _openDen(user2);

    // Rewards are activated
    vm.prank(owner);
    vault.registerReceiver(wBERADenManager, 2); // internally calls
↪   notifyRegisteredId

    // Deploy the mock of Vault that returns 1e18 on `allocateNewEmissions`
    // It also logs the amount when calling `transferAllocatedTokens`
    VaultMock vaultMock = new VaultMock();
    // Save the bytecode of the VaultMock to Vault
    vm.etch(address(vault), address(vaultMock).code);

    vm.warp(block.timestamp + 1 weeks);

    // Here happens the call to `_fetchRewards` and the rewardRate is set
    // We call it twice so both users have the same amount of active debt (due to
↪   interest accrued)
    vm.prank(user1);
    IDenManager(wBERADenManager).claimReward(user1);
```

```
    vm.prank(user2);
    IDenManager(wBERADenManager).claimReward(user2);

    // We check that the reward rate is correct -- distribute 1e18 tokens in 1 week
    uint256 rewardRate = IDenManager(wBERADenManager).rewardRate();
    assertEq(rewardRate, uint256(1e18 / uint(1 weeks)));

    // Half a week passes
    uint256 halfWeek = 1 weeks / 2;
    vm.warp(block.timestamp + halfWeek);

    // User1 claims reward
    vm.prank(user1);
    IDenManager(wBERADenManager).claimReward(user1);    // LOG: Amount claimed is
  ↪ 2.49999999999976799e17

    // The rest of the week passes
    vm.warp(block.timestamp + halfWeek);

    // Both users claim rewards
    vm.prank(user1);
    IDenManager(wBERADenManager).claimReward(user1);    // LOG: Amount claimed is
  ↪ 2.49999999999976799e17
    vm.prank(user2);
    IDenManager(wBERADenManager).claimReward(user2);    // LOG: Amount claimed is
  ↪ 4.99976029695736197e17
}
```

Results:

- User1 claimed rewards twice, totaling `2.49999999999976799e17` each time.
- User2 claimed rewards once, totaling `4.99976029695736197e17`.

This demonstrates that User1 received more rewards than User2 due to frequent updates, highlighting the flawed reward distribution system.

Tool used

Manual Review

Recommendation

To resolve this issue, rewards should be distributed based on the total active debt, including accrued interest. Similarly, users' reward calculations should use the debt balance with accrued interest applied.

This recommendation should be thoroughly tested to ensure it resolves the issue without introducing new risks.

Issue M-14: Interest Accrual on `DenManager` Will Be Incorrect Due to BrimeDen

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/198

The protocol has acknowledged this issue.

Summary

When global interest is accrued in `DenManager`, certain variables are miscalculated due to the unique condition of BrimeDen, which does not pay any interest. This results in the protocol collecting incorrect amounts of interest and locking some rewards.

Vulnerability Detail

Whenever a Den is updated within a `DenManager` instance, the `_accrueActiveInterests` function is called to update the global interest accrued since the last index update:

```
    // This function must be called any time the debt or the interest changes
    function _accrueActiveInterests() internal returns (uint256) {
        (uint256 currentInterestIndex, uint256 interestFactor) =
 ↪  _calculateInterestIndex();
        if (interestFactor > 0) {
            uint256 currentDebt = totalActiveDebt;
            uint256 activeInterests = Math.mulDiv(currentDebt, interestFactor,
 ↪  INTEREST_PRECISION);
>>          totalActiveDebt = currentDebt + activeInterests;
>>          interestPayable = interestPayable + activeInterests;
            activeInterestIndex = currentInterestIndex;
            lastActiveIndexUpdate = block.timestamp;
        }
        return currentInterestIndex;
    }
```

This function assumes that all Dens pay interest and uses `totalActiveDebt` to calculate the accrued interest. However, BrimeDen is a special contract that does not pay interest, meaning its debt should be excluded from the interest calculation.

Impact

When BrimeDen holds debt within a `DenManager`, the `_accrueActiveInterests` function calculates an inflated `activeInterests` amount because it incorrectly includes the BrimeDen's debt, which does not accrue interest.

This causes the following issues:

- The `interestPayable` variable becomes overstated, leading the protocol to collect more interest than it should.
- The `totalActiveDebt` variable is inflated, which results in locked rewards. Since rewards are distributed across all debt within the `DenManager` using `totalActiveDebt`, the inclusion of BrimeDen's non-paying debt leads to inaccurate reward distribution.

Tool used

Manual Review

Recommendation

To address this issue, the BrimeDen's debt should be subtracted from the `totalActiveDebt` before calculating accrued interest. Alternatively, the protocol could adopt a simpler solution by requiring BrimeDen to pay interest like other Dens.

Discussion

alex-beraborrow

Acknowledging, we had written the following in the Audit documentation:

```
Zero interest (we acknowledge that increases the real interest others have to pay,
↪  we will counter-effect this with lowering gross interests when the brimeDen
↪  debt over the total system debt increases)
```

Issue M-15: Massive MEV Opportunity for sunsetting scenario

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/215

The protocol has acknowledged this issue.

Summary

`startSunset` sets the configuration for a `denManager` to the following:

```
function startSunset() external onlyOwner {
    sunsetting = true;
    _accrueActiveInterests();
    interestRate = SUNSETTING_INTEREST_RATE;
    // accrual function doesn't update timestamp if interest was 0
    lastActiveIndexUpdate = block.timestamp;
    redemptionFeeFloor = 0;
    maxSystemDebt = 0;
}
```

This is reducing the redemption fee to zero

Which is opening up the system to Oracle Drift Arbitrage

Vulnerability Detail

Oracle Drift is the difference between the Oracle Reported Price and the asset Real Price (the price at which the asset is trading at in a Cex or Dex)

Redemptions necessitate having a minimum fee to protect against this type of risk free arbitrage which goes to the detriment of the Den being redeemed against

Impact

Setting the redemption fee to 0 and not raising it based on demand will open up to arbitraging the Den's Collateral

This can be done by either buying Nect from a pool and swapping it for the collateral

Or possibly can be done via a statistical arbitrage by minting from another Collateral and redeeming into the Collateral that is being shutdown

Any time the Price Oracle will under-report the value of the collateral, the arbitrage can be performed as the fee that was protecting against this will be removed

Tool used

Manual Review

Recommendation

If it's clear that people will get redeemed, you may just offer a peripheral system to trigger the redemption and socialize the gain to all other stakers I believe selling this MEV opportunity is probably better than giving it out to MEV bots at the detriment of real users

You could setup a redemption vault, that can trigger the proposal (has executor role to perform the sunsetting) and redeem 100% of the collateral atomically

The vault would then redistribute these profits

Basically you can force migrate positions which I think minimizes losses to the users

Discussion

santipu03

Marking a ”Won't Fix” as the actual fix will come in a future release

Issue M-16: Operative risk in raising the CCR without a Grace Period

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/221

The protocol has acknowledged this issue.

Summary

Raising the CCR at any time will allow people to borrow to drag the TCR close to RM and then trigger RM to liquidate people

Vulnerability Detail

The CCR is a critical value that helps the system perform liquidations for overly collateralized Dens

Any raise of the CCR can be sandwhiced as it will break the key invariant that no user action can trigger RM directly

While the invariant will be technically maintained, the economic value that can be generated by triggering RM "at will" is so high that adding this change necessarily requires thinking around the MEV it will generate

Impact

When the CCR is raised, liquidations could be triggered by an attacker that lowers the TCR close to the current CCR limit

If permissioneless execution is allowed by your governance contract, then this will not just be sandwiceable, but it will also be a guaranteed attack

Tool used

Manual Review

Recommendation

I'm thinking that slowly increasing the CCR change is ineffective agains this The only mitigation is to enforce a short period in which RM Liquidations cannot happen after the change has been done

As to allow people to change their CCR

The alternative is to effectively never change this And expect liquidations when you do

Discussion

santipu03

The fix PR for this issue has been closed given that the multisig changing the CCR already has a timelock. Also, the project stated that they'll use public comms to advise of any potential change to CCR, giving time to users to react before the change.

Thus, the issue should be marked as acknowledged

Issue M-17: Bex LP Token Pricing is subject to redemption skimming arbitrage

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/222

Summary

The `StableBexFeed`

prices the LP token in this way:

https://github.com/Beraborrowofficial/blockend/blob/9d133bb7a4a7b0a065931ada67c7319c5063e5d1/src/core/spotOracles/bex/StableBexFeed.sol#L142-L164

```
function _getTokensMinPrice(
    address[] memory tokens,
    address[] memory rateProviders,
    uint length
) internal view returns (uint) {
    uint minPrice;
    address minToken;

    for (uint i; i < length; ++i) {
        address minCandidate = tokens[i];
        IRateProvider rateProvider = IRateProvider(rateProviders[i]);
        uint minCandidatePrice = _calculateMinCandidatePrice(rateProvider,
↪   minCandidate);

        if (minCandidatePrice < minPrice || i == 0) {
            minToken = minCandidate;
            minPrice = minCandidatePrice;
        }
    }

    return minPrice;
}
```

This means that of the assets in the pool, we're taking the minimum reported price

In doing so we're opening up to an arbitrage whenever one of the two asset's oracle is under-reporting it's price due to Oracle Drift

Vulnerability Detail

Anytime one of the two asset price feeds underreports it's price, the redemption price of the LP token will be depressed since we're taking the minimum price out of the two assets

This makes redemptions highly likely to be profitable to any arbitrageur that is buying Nect at Parity

Impact

Due to the underpricing the LP token will be redeemed more than intended, causing depositors to lock in real losses (as they have to pay the highest of the prices for the LP instead of the lowest)

Tool used

Manual Review

Recommendation

Consider either having a very high redemption fee, or possibly taking the max of the LP price when pricing redemptions, while using the min when pricing the collateral

Discussion

GalloDaSballo

It seems like from my research that the LP token will be subject to a lot more arbitrage when the LP position is imbalanced I'm still looking into this

Issue M-18: Unfair redistribution when a Den is liquidated with bad debt

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/225

The protocol has acknowledged this issue.

Summary

When a Den is liquidated with bad debt, the remaining debt and collateral will be redistributed across the `DenManager` causing unfairness as some of the safest Dens will be the ones with a higher decrease in ICR.

Vulnerability Detail

When a Den is liquidated and it has bad debt (ICR < 100%), the remaining collateral and debt will be redistributed in the `DenManager` based on the collateral each Den currently has. This redistribution mechanism is unfair because it's just based on the Den's collateral and not the debt, causing some of the safest Dens to receive a greater impact due to the redistribution of debt.

Consider the following scenario:

- Bob and Alice want to borrow 100 NECT
- Bob decides to play safe and deposits 200 collateral, meaning **Bob's ICR will be 2**
- Alice prefers to play risky and deposits only 130 collateral, meaning **Alice's ICR will be 1.3**
- A liquidation with bad debt happens. 50 collateral and 80 debt have to be distributed between Bob and Alice.
- With the current redistribution system, their final balances will be:
  - Bob will have ~230.3 collateral and ~148.48 debt (**Bob's new ICR = ~1.55**)
  - Alice will have ~149.7 collateral and ~131.52 debt (**Alice's new ICR = ~1.13**)
- Bob's ICR has decreased from 2 to ~1.55, so it's a **22.5% decrease**
- Alice's ICR has decreased from 1.3 to ~1.13, so it's a **13.07% decrease**

Bob's ICR has decreased more than Alice's just because he preferred to play safe, which is best for the protocol.

Impact

When a Den is liquidated with bad debt, the Dens with more collateral (no matter their debt) will receive higher debt, meaning their ICR will be lowered more than riskier Dens with less collateral.

Tool used

Manual Review

Recommendation

To mitigate this issue, it's recommended to distribute the debt and collateral separately and based on each individual Den's debt and collateral. This way, Bob would have received the same debt as Alice but more collateral, which is fair behavior.

That said, we agree that a liquidation with bad debt is quite unlikely to happen (the protocol has a lot of measures to prevent it) and the fix for this issue would likely increase the complexity in code, which is probably the reason why Liquity hasn't changed this mechanism.

Issue M-19: Bad Debt Redistribution happening at the end of batch liquidation causes outsized value to Liquidator and higher than intended bad debt to troves receiving the bad debt

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/229

The protocol has acknowledged this issue.

Summary

This is a finding from liquity that I don't believe needs to be fixed but needs to be modelled and monitored

Basically the Bad Debt redistribution is happening at the end of the loop, this means that technically the remaining Dens are getting more bad debt than if you always redistribute before each liquidation

Vulnerability Detail

You can read more about it here by rvierdiiev:
https://github.com/code-423n4/2023-10-badger-findings/issues/36

These 2 are from Hyh and Stermi, whom reviewed eBTC https://github.com/GalloDaSballo/ebtc-cantina-latest/blob/main/md/cantinasec-review-badgerdao-42.md

https://github.com/GalloDaSballo/ebtc-cantina-latest/blob/main/md/cantinasec-review-badgerdao-43.md

Recommendation

Model and monitor whether bad debt being slightly higher than intended could cause a liquidation cascade or systematic risk to the system

Issue M-20: InfraredCollateralVault and permissioneless dust claim allows for self-liquidation

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/230

The protocol has acknowledged this issue.

Summary

`InfraredCollateralVault` values it's assets as follows: https://github.com/sherlock-audit/2024-11-beraborrow/blob/579ae86467dcc0452a6ced8aec34e392751f1c90/InfraredCollateralVault.sol#L170-L200

This includes `_convertToValue` which itself will call:

```
uint futureGrossEmission = infraredVault().earned(address(this), token);
```

Meaning the `totalAssets` is pricing in the rewards that the vault has yet to claim

Vulnerability Detail

Per https://github.com/sherlock-audit/2024-11-beraborrow/issues/159 we know that we can grief these rewards, causing a loss in the `totalAssets`

Impact

Due to this, we can:

- Create a healthy position, where some of the value comes off of pending unclaimed rewards
- Claim the rewards on behalf of the vault, causing a loss to `totalAssets`
- Liquidate our position, causing in bad debt to the system

Tool used

Manual Review

Recommendation

I believe that you must prevent rewards from being claimable from external parties, in the case in which they were, those rewards should be harvested on the next deposit as to prevent `totalAssets` from ever being reduced

Discussion

alex-beraborrow

Will fix, same root issue as:
https://github.com/sherlock-audit/2024-11-beraborrow/issues/159

dmitriia

In addition to Alex's thoughts: a frequent enough keeper bot claiming can be run, so there be not enough incentives to capture the accrual. Griefing in #159 can have various incentives, but here it's this part of the yield only, so it can be just kept low in value, so both bad debt creation and yield capture be too low in size to be acted on

GalloDaSballo

The Issue was not mitigated in my opinion, I shared a private gist with the team

alex-beraborrow

https://github.com/sherlock-audit/2024-11-beraborrow/issues/159
allowbreak #issuecomment-2592662266

Issue M-21: Vesting `collSurplusAmount` after `offset` can cause Price Per Share value and open up to losses to LSP depositors

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/235

The protocol has acknowledged this issue.

Summary

Liquidations are handled in the LSP via `offset` which will compute a spot `collSurplus`and vest it over a vesting schedule.

However this estimation will be incorrect at the next oracle update, and in half the cases the price could decrease, causing a reduction in `totalAssets` and a loss to depositors

Vulnerability Detail

Let's imagine for the sake of argument that we have the "current price" and the "next price" And just to keep it simple assume I can chose to push the "next price"

The liquidation will happen with the current price The offset will be computed

If the next price depresses the value of collaterals, this could open up to an arbitrage as we would now be able to vest those while also getting a discount on the PPFS

Impact

As flagged we have a ton of MEV opportunities here

Another fairly big one is the fact that `offset` will start vesting the surplus

The goal being that the PPFS will not change instantly

However, this makes it so that once a new oracle update happens, and the update reprices the collateral negatively, the PPFS of the LSP will decrease

This will raise the likelyhood that arbitrageurs will gain value by staking AFTER liquidations rather than before them

If the system were to use Pyth as in the current scope, this would most likely guarantee an arbitrage instead of just making it likely

Code Snippet

Tool used

Manual Review

Recommendation

I believe that the whole logic of pricing assets need to be rethought

Generally speaking there may be better ways to price and auction collateral to be converted back to nect

And most importantly the PPFS of the LSP should never decrease

Discussion

alex-beraborrow

By doing the Issue #182 fix, this should stop being a problem. No MEV opportunity after the liquidation happened.

santipu03

Marking a ”Won't Fix” as the actual fix will come in a future release (same as issue #182)

Issue M-22: BPT Tokens cannot be Flashloaned as they can otherwise open up to risk free arbitrage by withdrawing and redepositing

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/240

Summary

Because of how Stable BPTs work, where they assume that both tokens have the same value, and punish imbalances by granting less tokens, while rewarding rebalancing the pool

Flashloaning can be abused as a way to skim tokens, rebalance the pool and gain the excess value

This means that all BPT depositors are willingly giving away their "premium" to flashloan callers, causing a loss to them

Vulnerability Detail

It's worth noting that passive LPing already shares this issue

Flashloaning just makes the arbitrage easier to execute

Code Snippet

We demonstrate the arbitrage by providing Echidna with the following handlers:

- A way to LP
- A test `step_1_arb` that withdraws 1e18 of LP token and then deposits single sided to obtain back 1e18 tokens, it then flags a failure anytime the cost of the single sided LPIng is cheaper than the value received by withdrawing both assets

```
// forge test --match-test test_step_1_arb_0 --fork-url
↪  https://eth-mainnet.g.alchemy.com/v2/KEY -vvvv
function test_step_1_arb_0() public {
    balancer_supply(0,417482529357807651200);
    step_1_arb();
}
```

Tool used

Manual Review

Recommendation

I believe that BPT tokens cannot be used with the system unless you disable Redemptions and Flashloaning As well as any additional mechanism that allows an actor to receive the BPT underlying

Issue M-23: Locked Funds in `ValidatorPool` Due to Sunsetting of a Collateral Token

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/241

Summary

When a collateral token is sunsetted and subsequently removed from the `LiquidStabilityPool` (LSP), it can lead to funds being permanently locked in the `ValidatorPool` if recent liquidations involved the token.

Vulnerability Detail

In Beraborrow, during a liquidation, the compensation tokens are distributed among the liquidator, the `sNECT` gauge, and the `ValidatorPool`. When funds reach the `ValidatorPool`, the `distribute` function can be called anytime to allocate these tokens pro-rata to the whitelisted validators.

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

        // ...

        for (i = 0; i < collateralTokens.length; i++) {
            tokenBalance = IERC20(collateralTokens[i]).balanceOf(address(this));
            if (tokenBalance > 0) {
                for (uint j; j < validatorCount; j++) {
                    address validator = memValidators[j];
                    uint amount = (tokenBalance * memShares[j]) / BASIS_POINT;
                    if (amount > 0) {
>>                      IERC20(collateralTokens[i]).safeTransfer(validator, amount);
                    }
                }
            }
        }
    }
```

The distribute function retrieves the array of collateral tokens from the LSP and iterates through them to distribute the funds to all validators.

However, if a collateral token is sunsetted and the 180-day grace period elapses, the token is removed from the array. This removal causes any tokens of that type remaining in the `ValidatorPool` to become permanently locked, as the `distribute` function will no longer process them.

Scenario Example

1. A collateral token is sunsetted.
2. Time passes.
3. Liquidations occur on the sunsetted `DenManager`.
4. More time passes.
5. The sunsetted collateral is removed from the LSP, excluding it from the list of collateral tokens.
6. The tokens resulting from liquidations that remain in `ValidatorPool` cannot be distributed since the `distribute` function no longer recognizes the token.

This issue arises if the `distribute` function is not invoked between the liquidations and the removal of the collateral token from the LSP.

Impact

The funds corresponding to the removed collateral token will remain locked in the `ValidatorPool` indefinitely.

Tool used

Manual Review

Recommendation

Introduce a recovery function in `ValidatorPool` to allow the retrieval of tokens that are neither NECT nor listed as collateral tokens in the LSP.

Issue M-24: LeverageRouter can be unavailable due to the mixture of inexact calculations and exact controls

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/243

Summary

LeverageRouter's `_handleRepayment()` derive amount needed for `deposit()` with `previewMint()` and requests that `collVault` shares minted exactly cover the needed margin. This will not be true whenever rounding truncates the output, leading to unavailability of all LeverageRouter's user facing functionality.

Vulnerability Detail

ERC4626's `deposit()` uses `previewDeposit()`, while `mint()` uses `previewMint()`:

ERC4626Upgradeable.sol#L193-L217

```
    /** @dev See {IERC4626-deposit}. */
    function deposit(uint256 assets, address receiver) public virtual returns
 ↪  (uint256) {
        uint256 maxAssets = maxDeposit(receiver);
        if (assets > maxAssets) {
            revert ERC4626ExceededMaxDeposit(receiver, assets, maxAssets);
        }

>>      uint256 shares = previewDeposit(assets);
>>      _deposit(_msgSender(), receiver, assets, shares);

        return shares;
    }

    /** @dev See {IERC4626-mint}. */
    function mint(uint256 shares, address receiver) public virtual returns
 ↪  (uint256) {
        uint256 maxShares = maxMint(receiver);
        if (shares > maxShares) {
            revert ERC4626ExceededMaxMint(receiver, shares, maxShares);
        }

>>      uint256 assets = previewMint(shares);
>>      _deposit(_msgSender(), receiver, assets, shares);
```

```
        return assets;
    }
```

Which means the deposit and mint use different rounding:

ERC4626Upgradeable.sol#L173-L181

```
    /** @dev See {IERC4626-previewDeposit}. */
    function previewDeposit(uint256 assets) public view virtual returns (uint256) {
>>      return _convertToShares(assets, Math.Rounding.Floor);
    }

    /** @dev See {IERC4626-previewMint}. */
    function previewMint(uint256 shares) public view virtual returns (uint256) {
>>      return _convertToAssets(shares, Math.Rounding.Ceil);
    }
```

This way current logic is prone to function unavailability due to rounding induced `marginMinted < missingMargin` state occurring when the numbers are so that `_convertToShares(_convertToAssets(missingMargin, Math.Rounding.Ceil), Math.Rounding.Floor) < missingMargin`:

LeverageRouter.sol#L208-L240

```
    function _handleRepayment(
        ...
    ) internal {
        uint payBackAmount = amount + fee;
        if (payBackAmount > collMinted) {
            uint missingMargin = payBackAmount - collMinted;
            uint marginMinted = _transferFromMissingMargin(asset, account,
↪   missingMargin, collVault);
>>          require(marginMinted >= missingMargin, "Leverage: marginMinted <
↪   missingMargin");
            IERC20(collVault).approve(msg.sender, payBackAmount);
        } else {
            // Since MCR > 100%, unlikely to enter this branch
            IERC20(collVault).approve(msg.sender, payBackAmount);
            IInfraredCollateralVault(collVault).redeem(collMinted - payBackAmount,
↪   account, address(this));
        }
    }

    /// @dev Calculates missing margin in assets, transfers them to the contract
↪   and wraps them to Collateral Vault
    function _transferFromMissingMargin(address asset, address account, uint
↪   missingMargin, address collVault) private returns (uint marginMinted) {
>>      uint requiredAssets =
↪   IInfraredCollateralVault(collVault).previewMint(missingMargin);
        IERC20(asset).safeTransferFrom(account, address(this), requiredAssets);
```

```
        marginMinted = _wrapAssetToCollVault(collVault, asset, requiredAssets);

    }

    function _wrapAssetToCollVault(address collVault, address asset, uint amount)
 ↪  private returns (uint collMinted) {
        IERC20(asset).approve(collVault, amount);
>>      collMinted = IInfraredCollateralVault(collVault).deposit(amount,
 ↪  address(this));
    }
```

Both LeverageRouter operations, `automaticLoopingOpenDen()` and `automaticLoopingAddCollateral()`, invoke `_processFlashLoan()` -> `_handleRepayment()`:

LeverageRouter.sol#L146-L148

```
    uint collMinted = _swapAndWrap(params.dexAggregator, asset, collVault,
 ↪  params.denParams.debtAmount);
    _handleRepayment(amount, fee, collMinted, asset, account, collVault);
}
```

Impact

`automaticLoopingOpenDen()` and `automaticLoopingAddCollateral()` will be unavailable whenever `_convertToShares(_convertToAssets(missingMargin, Math.Rounding.Ceil), Math.Rounding.Floor) < missingMargin`, which will take place all the time `missingMargin` happen to be not a round number.

Recommendation

Consider calling `IInfraredCollateralVault(collVault).mint(missingMargin, address(this))` instead of using `_wrapAssetToCollVault()`, so the computations be exact.

Discussion

dmitriia

Ok (PR92)

Issue M-25: LeverageRouter's `automaticLoopingAddCollateral()` causes losses for a user when swap proceeds exceed flash loan amount due

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/244

Summary

When LeverageRouter's `automaticLoopingAddCollateral()` invoked flash loan is run for a part of the collateral needed for new debt, with the other part coming from decreasing ICR of the position used (which is handy as it unifies two operations, so is a viable use case), current logic incurs losses to a user by redundant wrapping and redeeming.

Say Bob can set `params.debtAmount` to `NECT` amount equivalent to `1500 ColToken`, while getting flash loan for `collAssetsToDeposit = 500 ColToken` only, using the existing collateral to cover the collateralization of the rest part of the new debt by lowering the ICR of the position.

Vulnerability Detail

The case of `payBackAmount <= collMinted`, contrary to L223 comment, is not tied to `MCR` as any user having excess collateralization can request minting more debt than flash loaned part of the collateral allows, by driving their ICR down in the process. I.e. as ICR is being adjusted both excess and deficit `collMinted` cases are valid and not improbable.

`_swapAndWrap()` wraps all the collateral base asset received from the swapping:

LeverageRouter.sol#L190-L206

```
    function _swapAndWrap(DexAggregatorParams memory params, address asset, address
↪   collVault, uint nectAmount) private returns (uint collMinted) {
        uint prevCollBalance = IERC20(asset).balanceOf(address(this));

        // Referral code could be hardcoded
        require(params.dexCalldata.getSelector() == IOBRouter.swap.selector,
↪   "Leverage: Invalid dex calldata");
        nect.approve(obRouter, nectAmount);
        (bool success, bytes memory retData) = obRouter.call(params.dexCalldata);

        if (!success) {
            retData.bubbleUpRevert();
        }
```

```
        uint collAssetReceived = IERC20(asset).balanceOf(address(this)) -
  ↪ prevCollBalance;
        require(collAssetReceived >= params.collOutputMin, "Leverage: collReceived
  ↪ < collOutputMin");

>>      collMinted = _wrapAssetToCollVault(collVault, asset, collAssetReceived);
    }
```

And then `_handleRepayment()` unwraps it back:

LeverageRouter.sol#L208-L227

```
    function _handleRepayment(
        ...
    ) internal {
        uint payBackAmount = amount + fee;
        if (payBackAmount > collMinted) {
            uint missingMargin = payBackAmount - collMinted;
            uint marginMinted = _transferFromMissingMargin(asset, account,
  ↪ missingMargin, collVault);
            require(marginMinted >= missingMargin, "Leverage: marginMinted <
  ↪ missingMargin");
            IERC20(collVault).approve(msg.sender, payBackAmount);
        } else {
            // Since MCR > 100%, unlikely to enter this branch
            IERC20(collVault).approve(msg.sender, payBackAmount);
>>          IInfraredCollateralVault(collVault).redeem(collMinted - payBackAmount,
  ↪ account, address(this));
        }
    }
```

Impact

This will cause losses for the user due to back and forth rounding (amount to shares) and InfraredCollateralVault fees (entry, exit).

Recommendation

One way to fix this is to add wrapping slippage control, another is to remove this excessive wrapping altogether. The latter is somewhat preferred since `_swapAndWrap()` and `_handleRepayment()` aren't used elsewhere. Consider unifying these functions, e.g.:

LeverageRouter.sol#L146-L149

```
-       uint collMinted = _swapAndWrap(params.dexAggregator, asset, collVault,
  ↪ params.denParams.debtAmount);
-       _handleRepayment(amount, fee, collMinted, asset, account, collVault);
```

```
+       _swapWrapRepay(amount, fee, account, params.dexAggregator, asset,
↪  collVault, params.denParams.debtAmount)
```

LeverageRouter.sol#L190-L206

```
-   function _swapAndWrap(DexAggregatorParams memory params, address asset, address
↪   collVault, uint nectAmount) private returns (uint collMinted) {
+   function _swapWrapRepay(uint amount, uint fee, address account,
↪   DexAggregatorParams memory params, address asset, address collVault, uint
↪   nectAmount) private {
        ...  // exactly the same as _swapAndWrap() until the last line, which is
↪   replaced as:
-       collMinted = _wrapAssetToCollVault(collVault, asset, collAssetReceived);
+       uint payBackAmount = amount + fee;
+       uint requiredAssets =
↪   IInfraredCollateralVault(collVault).previewMint(payBackAmount);
+       uint usedAssets;
+       if (requiredAssets > collAssetReceived) {
+           IERC20(asset).safeTransferFrom(account, address(this), requiredAssets -
↪   collAssetReceived);
+           IERC20(asset).approve(collVault, requiredAssets);
+           usedAssets = IInfraredCollateralVault(collVault).mint(payBackAmount,
↪   address(this));
+       } else {
+           IERC20(asset).approve(collVault, requiredAssets);
+           usedAssets = IInfraredCollateralVault(collVault).mint(payBackAmount,
↪   address(this));
+           IERC20(asset).safeTransfer(account, collAssetReceived - usedAssets);
+       }
+       require(requiredAssets >= usedAssets, "Leverage: requiredAssets <
↪   usedAssets");
+       IERC20(collVault).approve(msg.sender, payBackAmount);
+   }
```

Discussion

alex-beraborrow

This also applies for `automaticLoopingOpenDen` and includes https://github.com/sherlock-audit/2024-11-beraborrow/issues/243 recommendation right?

dmitriia

> This also applies for `automaticLoopingOpenDen` and includes #243 recommendation right?

`automaticLoopingOpenDen` opens a new Den afresh, so the situation described can happen only if the swap pool used being highly misbalanced vs protocol Oracle reported price, i.e. when swap returned collateral covers the new debt (also it's full debt since the position is new) by more than `100\%`. Otherwise, while `MCR > 100\%` there should be no such situation. However, the code suggested is somewhat shorter and does fix #243, so it worth updating `\_processFlashLoan()` logic as described, covering both adjustment and opening.

And so yes, this includes #243 recommendation fully.

`\_handleRepayment()`, `\_transferFromMissingMargin()` and `\_wrapAssetToCollVault()` look to become unused after that and can be removed if there are no other ideas there.

dmitriia

Ok (PR92)

Issue M-26: Broken extra asset can block all LSP and InfraredCollateralVault ERC4626 operations, freezing all the assets there

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/247

The protocol has acknowledged this issue.

Summary

LSP and InfraredCollateralVault ERC4626 operations (mint, deposit, withdraw, redeem) are reliant on `totalAssets()`, which unconditionally cycle through extra assets list. It's not possible to remove an extra asset with positive balance in LSP and there is no reward token removal functionality in InfraredCollateralVault.

Vulnerability Detail

In the LiquidStabilityPool case since it will be impossible to reduce `$.balance[token]` due to token reverting transfers it will not be possible to remove token from the `extraAssets` list with `removeExtraAsset()`:

LiquidStabilityPool.sol#L684-L692

```
    function removeExtraAsset(address token) external onlyOwner {
        ILiquidStabilityPool.LSPStorage storage $ = _getLSPStorage();

>>      require($.balance[token] == 0, "LSP: token has balance");
        require($.emissionSchedule[token].unlockTimestamp() < block.timestamp,
 ↪  "LSP: token is vesting");
        require($.extraAssets.remove(token), "LSP: token is not withdrawable");

        emit ExtraAssetRemoved(token);
    }
```

In the InfraredCollateralVault case there is only list addition via `_addRewardedToken()`, but no removal.

In both cases having inoperable token in the list will block all the `totalAssets()` calls and thus all base implementations of ERC4626 operations (mint, deposit, withdraw, redeem), which are used in both Vaults.

Impact

Pausing or malfunction of an asset from the LSP's `extraAssets` or InfraredCollateralVault's `rewardedTokens` will block all the relying ERC4626 operations. In the worst case of the token remaining bricked for good it will permanently freeze all the assets in LSP and InfraredCollateralVault, sunsetting the protocol.

While the impact is critical, the probability of a prolonged/permanent freeze is very low, so setting the severity to be medium.

Recommendation

Consider introducing an option to LSP's `removeExtraAsset()` to remove an extra asset with positive balance to handle such a case. Consider introducing the similar force removal of an asset from `rewardedTokens` in InfraredCollateralVault.

In both cases `rebalance()` can be used to swap this token away later if/when it unfreezes without it being on the list.

Discussion

alex-beraborrow

Rebalance can set it to 0 easily with the swapper.

dmitriia

> Rebalance can set it to 0 easily with the swapper.

No, if asset reverts the transactions.

Rebalance requires asset to be operable, otherwise it can't do anything:

```
// Perform the swap using the swapper contract
IERC20(p.sentCurrency).safeTransfer(p.swapper, p.sentAmount);  // @audit: reverts
...
\$.balance[p.sentCurrency] -= sent;  // @audit: can't happen
```

dmitriia

Acknowledged

Issue M-27: PriceFeed's Chainlink logic doesn't fail on negative and boundary prices, potentially overpricing the assets and allowing bad debt creation

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/251

Summary

PriceFeed explicitly converts Chainlink returned `int256 answer` to `uint256`, so on seeing a negative price it will produce huge positive price instead of failing.

Also, there is no min and max answer control, so a known issue with hitting min and max boundaries can happen as well.

Both scenarios have low probability, but allow for drastic overvaluation and massive bad debt creation.

Vulnerability Detail

Explicit `int` to `uint` conversion is used in PriceFeed.sol L221, L274, L275:

PriceFeed.sol#L209-L231

```
    function _processFeedResponses(
        ...
    ) internal view returns (uint256) {
        uint8 decimals = oracle.decimals;
        bool isValidResponse = _isFeedWorking(_currResponse, _prevResponse) &&
            !_isPriceStale(_currResponse.timestamp, oracle.heartbeat) &&
            !_isPriceChangeAboveMaxDeviation(_currResponse, _prevResponse,
↪   decimals);

        if (isValidResponse) {
>>          uint256 scaledPrice =
↪   _scalePriceByDigits(uint256(_currResponse.answer), decimals);
            if (oracle.sharePriceSignature != 0) {
                (bool success, bytes memory returnData) =
↪   _token.staticcall(abi.encode(oracle.sharePriceSignature));
                require(success, "Share price not available");
                scaledPrice = (scaledPrice * abi.decode(returnData, (uint256))) /
↪   (10 ** oracle.sharePriceDecimals);
            }
            if (oracle.isEthIndexed) {
```

```
            // Oracle returns against 'native token' (BERA) price, need to
↪   convert to USD
            scaledPrice = _calcBeraPrice(scaledPrice);
        }
        return scaledPrice;
```

PriceFeed.sol#L269-L275

```
    function _isPriceChangeAboveMaxDeviation(
        FeedResponse memory _currResponse,
        FeedResponse memory _prevResponse,
        uint8 decimals
    ) internal pure returns (bool) {
>>      uint256 currentScaledPrice =
↪   _scalePriceByDigits(uint256(_currResponse.answer), decimals);
>>      uint256 prevScaledPrice =
↪   _scalePriceByDigits(uint256(_prevResponse.answer), decimals);
```

Those conversions won't fail on negative numbers, silently producing huge positive answers instead.

Also, there is no min and max price control, so those answers also will be passed over, allowing misvaluation:

PriceFeed.sol#L260-L267

```
function _isValidResponse(FeedResponse memory _response) internal view returns
↪   (bool) {
    return
        (_response.success) &&
        (_response.roundId != 0) &&
        (_response.timestamp != 0) &&
        (_response.timestamp <= block.timestamp) &&
        (_response.answer != 0);
}
```

Both scenarios can end up overpricing the collateral:

DenManager.sol#L390-L396

```
    function fetchPrice() public view returns (uint256) {
        IPriceFeed _priceFeed = priceFeed;
        if (address(_priceFeed) == address(0)) {
            _priceFeed = IPriceFeed(BERABORROW_CORE.priceFeed());
        }
>>      return _priceFeed.fetchPrice(address(collateralToken));
    }
```

Impact

Chainlink malfunctions of returning negative or boundary price induces overvaluation and this way allows bad debt creation up to protocol insolvency.

While impact itself here is critical, the probability of such answers can be higher (low) than usual (very low) since Berachain Oracle setup is new.

Recommendation

Consider requiring answer to be positive, e.g. with `_response.answer > 0` in `_isValidResponse()`:

PriceFeed.sol#L260-L267

```
    function _isValidResponse(FeedResponse memory _response) internal view returns
 ↪  (bool) {
        return
            (_response.success) &&
            (_response.roundId != 0) &&
            (_response.timestamp != 0) &&
            (_response.timestamp <= block.timestamp) &&
-           (_response.answer != 0);
+           (_response.answer > 0);
    }
```

Also, consider adding min and max limits control there. This needs to be done inclusively, i.e. with failing on the boundaries, as oracle will continue to report boundary values in the case of the breach.

Discussion

dmitriia

Oracle provider boundary price behavior needs to be examined: what feed will do if price drops sharply, e.g. will last in-range price be reported or there be no such truncation?

If there be some kind of behavior switching when price is deemed out of bounds, as described in the discussions linked for Chainlink case, then this has to be controlled for, how exactly depends on the behavior details

Issue L-1: QA - Base Rate Decay may be slower than intended - Chaduke

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/160

Notes

This is an already known finding from ETHOS (Liquity Fork)

Impact

`_calcDecayedBaseRate` calls `_minutesPassedSinceLastFeeOp` which rounds down by up to 1 minute - 1

https://github.com/sherlock-audit/2024-11-beraborrow/blob/390336070b5ed6ff061e3a7742de71092092ee66/blockend/src/core/DenManager.sol#L616-L630

```
// Update the last fee operation time only if time passed >= decay interval. This
↪ prevents base rate griefing.
function _updateLastFeeOpTime() internal {
    uint256 timePassed = block.timestamp - lastFeeOperationTime;
    if (timePassed >= SECONDS_IN_ONE_MINUTE) {
        lastFeeOperationTime = block.timestamp;
        emit LastFeeOpTimeUpdated(block.timestamp);
    }
}

function _calcDecayedBaseRate() internal view returns (uint256) {
    uint256 minutesPassed = (block.timestamp - lastFeeOperationTime) /
↪   SECONDS_IN_ONE_MINUTE;
    uint256 decayFactor = BeraborrowMath._decPow(minuteDecayFactor, minutesPassed);

    return (baseRate * decayFactor) / DECIMAL_PRECISION;
}
```

This, in conjunction with the logic `_updateLastFeeOpTime`

Will make the decay factor decay slower than intended

Additional Readings

This finding was found in the ETHOS contest by Chaduke:
https://github.com/code-423n4/2023-02-ethos-findings/issues/33

Discussion

alex-beraborrow

Agree

Issue L-2: Economic Considerations for Beraborrow as well as Liquity V1 Forks

Source: https://github.com/sherlock-audit/2024-11-beraborrow/issues/164

Economic Considerations

It's important to consider that some assets may be massively more volatile than others

This means that some collaterals may have a lower total Dollar value, but may contribute an outsized amount to the risk of the system

Generally speaking for tail assets, the bigger question is whether they are even worth adding

Especially for a system like Liquity V1

Borrow Limits

A key tool that can be used to reduce risk are borrow limits

Enforcing a limit on riskier assets ensures that you cannot overly damage the system

IMO for risky assets that can be deprecated, a limit on total borrows (or a limit on deposits) is necessary

Cross Collateral Redemptions and Arbitrage

If every redemption has the same fee Then the collateral with the highest Oracle Deviation will on average leak the most value

This is due to Oracle Drift, the difference between the real price of an asset and the price reported by the oracle

Whenever sufficient drift is present, the system will leak value

This should on average happen more often for assets that are more volatile and that have a higher deviation threshold

Mitigation

Redemptions will have to have a base fee that matches the highest deviation threshold (scaled by decimals)

This will ensure that the base redemption fee will not automatically leak value against the oracle

Additional Research done for Ethos (another Liquity Fork)

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.