SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	RealWagmi
Prepared by:	Sherlock
Lead Security Expert:	0xDetermination
Dates Audited:	February 23 - February 27, 2024
Prepared on:	March 18, 2024

SHERLOCK

Introduction

Unlock the power of DeFi with Wagmi - an all-in-one platform for trading, liquidity provision, swapping, and yield strategy generation.

Scope

Repository: RealWagmi/wagmi-leverage

Branch: main

Commit: ca5e13121e2612c3961e51ed0b976ee8e5bae471

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
2	1

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Fees aren't distributed properly for positions with multiple lenders, causing loss of funds for lenders

Source:
https://github.com/sherlock-audit/2024-02-leverage-contracts-judging/issues/41

Found by

0xDetermination, zraxx

Summary

Fees distributed are calculated according to a lender's amount lent divided by the total amount lent, which causes more recent lenders to steal fees from older lenders.

Vulnerability Detail

The fees distributed to each lender are determined by the following calculation (https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L546-L549):

```
uint256 feesAmt = FullMath.mulDiv(feesOwed, cache.holdTokenDebt,
↪  borrowedAmount); //fees owed multiplied by the individual amount lent,
↪  divided by the total amount lent
...
loansFeesInfo[creditor][cache.holdToken] += feesAmt;
harvestedAmt += feesAmt;
```

The above is from `harvest()`; `repay()` calculates the fees similarly. Because `borrow()` doesn't distribute fees, the following scenario will occur when a borrower increases an existing position:

1. Borrower has an existing position with fees not yet collected by the lenders.
2. Borrower increases the position with a loan from a new lender.
3. `harvest()` or `repay()` is called, and the new lender is credited with some of the previous fees earned by the other lenders due to the fees calculation. Other lenders lose fees.

This scenario can naturally occur during the normal functioning of the protocol, or a borrower/attacker with a position with a large amount of uncollected fees can maliciously open a proportionally large loan with an attacker to steal most of the fees.

Also note that ANY UDPATE ISSUE? LOW PRIO

Impact

Loss of funds for lenders, potential for borrowers to steal fees.

Code Snippet

https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L546-L549

Tool used

Manual Review

Recommendation

A potential fix is to harvest fees in the borrow() function; the scenario above will no longer be possible.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/RealWagmi/wagmi-leverage/commit/84416fcedfcc7eb062917bdc69f919bba9d3c0b7.

fann95

Yes, the problem existed and is associated with the same error as #39. This issue is related to an erroneous scheme for accumulating fees and affected almost all functions in the contract, so the PR turned out to be quite large.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid: high(1)

nevillehuang

See comments here

0xDetermination

Fix looks good, a new internal function `_harvest()` has been added and is called when new borrows are added to an existing position.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Entrance fees are distributed wrongly in loans with multiple lenders

Source:
https://github.com/sherlock-audit/2024-02-leverage-contracts-judging/issues/39

Found by

0xDetermination

Summary

Entrance fees are distributed improperly, some lenders are likely to lose some portion of their entrance fees. Also, calling `updateHoldTokenEntranceFee()` can cause improper entrance fee distribution in loans with multiple lenders.

Vulnerability Detail

Note that entrance fees are added to the borrower's `feesOwed` when borrowing:

```
borrowing.feesOwed += entranceFee;
```

Also note that the fees distributed to each lender are determined by the following calculation (https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L546-L549):

```
uint256 feesAmt = FullMath.mulDiv(feesOwed, cache.holdTokenDebt,
 ↪  borrowedAmount); //fees owed multiplied by the individual amount lent,
 ↪  divided by the total amount lent
...
loansFeesInfo[creditor][cache.holdToken] += feesAmt;
harvestedAmt += feesAmt;
```

This is a problem because the entrance fees will be distributed among all lenders instead of credited to each lender. Example:

1. A borrower takes a loan of 100 tokens from a lender and pays an entrance fee of 10 tokens.
2. After some time, the lender harvests fees and fees are set to zero. (This step could be frontrunning the below step.)
3. The borrower immediately takes out another loan of 100 tokens and pays and entrance fee of 10 tokens.

4. When fees are harvested again, due to the calculation in the code block above, 5 tokens of the entrance fee go to the first lender and 5 tokens go to the second lender. The first lender has collected 15 tokens of entrance fees, while the second lender has collected only 5- despite both loans having the same borrowed amount.

Furthermore, if the entrance fee is increased then new lenders will lose part of their entrance fee. Example:

1. A borrower takes a loan of 100 tokens from a lender and pays an entrance fee of 10 tokens.
2. The entrance fee is increased.
3. The borrower increases the position by taking a loan of 100 tokens from a new lender, and pays an entrance fee of 20 tokens.
4. `harvest()` is called, and both lenders receive 15 tokens out of the total 30 tokens paid as entrance fees. This is wrong since the first lender should receive 10 and the second lender should receive 20.

Impact

Lenders are likely to lose entrance fees.

Code Snippet

https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L1036
https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L546-L549

Tool used

Manual Review

Recommendation

Could add the entrance fee directly to the lender's fees balance instead of adding it to feesOwed, and then track the entrance fee in the loan data to be used in min fee enforcement calculations.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/RealWagmi/wagmi-leverage/commit/84416fcedfcc7eb062917bdc69f919bba9d3c0b7.

fann95

Yes, the problem existed and is associated with the same error as #41. This issue is related to an erroneous scheme for accumulating fees and affected almost all functions in the contract, so the PR turned out to be quite large.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid; high(1)

nevillehuang

@fann95 Is the root cause the same as #41?

fann95

I think so since it was assumed that the entrance fee would be distributed the same way as the fees for borrowing.

nevillehuang

@fann95 Can you take a look at this comment and let me know your thoughts

fann95

> Can you take a look at this comment and let me know your thoughts

done

nevillehuang

See comments here

zrax-x

Escalate

This should be a duplicate of issue 41, or be of Medium severity.

Here are my two reasons.

Firstly, all of these issues (issue 41, issue 16 and this one) discuss the distribution of fees. In both issue 41 and issue 16, it was mentioned that the fees are not distributed in the function `borrow`, causing some lenders to lose fees. This issue talks about the same fee distribution problem, except it focuses on the entrance fees. In that respect, it's a Duplicate.

Secondly, entrance fees are a fraction of all fees, only 0.1% is charged by default and can be set to 0. Therefore, its impact is obviously not as serious as that mentioned in issue 41. I consider this to be a Medium severity issue.

sherlock-admin2

> Escalate
>
> This should be a duplicate of issue 41, or be of Medium severity.
>
> Here are my two reasons.
>
> Firstly, all of these issues (issue 41, issue 16 and this one) discuss the distribution of fees. In both issue 41 and issue 16, it was mentioned that the fees are not distributed in the function `borrow`, causing some lenders to lose fees. This issue talks about the same fee distribution problem, except it focuses on the entrance fees. In that respect, it's a Duplicate.
>
> Secondly, entrance fees are a fraction of all fees, only 0.1% is charged by default and can be set to 0. Therefore, its impact is obviously not as serious as that mentioned in issue 41. I consider this to be a Medium severity issue.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

qmdddd

Escalate

See above.

sherlock-admin2

> Escalate
>
> See above.

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.

0xDetermination

I disagree with the escalation's first point for the following reasons:

1. The root cause/fix for this issue is distinct from the root cause/fix for 41 (these fixes are implementing the recommendations). Fixing one issue won't fix the other.
2. Entrance fees are a different type of fees than fees accumulated over time.

As for the second point, I agree that this may be a borderline M, but I lean more towards H for the following reasons:

1. Entrance fees can be set up to

   Would like to see how @Czar102 judges this issue, since I do think the severity may be borderline.

   nevillehuang

   @zrax-x Can you provide code logic to prove the root cause is similar? If not I think it should remain not as duplicates given it involves different fee types.

   Additionally, I believe medium severity could be more appropriate, but based on what @fann95 has highlighted, the impact is widespread throughout the whole system. @fann95 Could you shed some light on the potential impact it could have and does it justify high severity?

   zrax-x

   I believe High severity is not appropriate.

   The entry fee rate defaults to 0.1% and entry fee is determined when borrowing (i.e. it does not increase over time), so its amount is limited.

   At the same time, in order to steal the entry fees, the attacker will have to pay fees to the platform (as implemented in the function _pickUpPlatformFees), which accounts for 20% of the interest fees. So the attack cost is very high.

   ```
   /**
    * @dev Platform fees in basis points.
    * 2000 BP represents a 20% fee on the daily rate.
    */
   uint256 public platformFeesBP = 2000;
   ```

   In summary, this issue will indeed cause some lenders to lose part of entry fees, but its impact is limited. I maintain it is a Medium severity issue.

   0xDetermination

   I think an impact that might make this issue H without needing to consider conditions like entrance fee settings is the 'net loss built up over time' example in my previous comment (edited to add the example).

Addressing @zrax-x's point about platform fees- true, there is a maximum/default 20% platform fee although it can also be set lower. If we assume slippage is negligible the attack would still be profitable, but I agree that this would reduce the profitability.

Czar102

@zrax-x @qmdddd can you follow up on @nevillehuang's question?

> Can you provide code logic to prove the root cause is similar?

Just want to have clarity on the duplication status before considering the severity. Do you agree that this issue shouldn't be a duplicate of #41?

zrax-x

@Czar102 I now believe this can be a distinct issue, although it also pertains to fee distribution, the distinction lies in the calculation methods for the fees.

Czar102

@zrax-x @qmdddd @0xDetermination what is the optimal attacker's strategy to minimize their fees? How much does the lender lose in that scenario?

0xDetermination

@Czar102 @zrax-x I looked into it more and I think the minimizing fee attack is actually not profitable unless the attacker is frontrunning the admin increasing entrance fees or is colluding with another lender that the attacker legitimately wants to borrow from. So I think the highest impact for this issue may be the 'net loss' scenario as described in my earlier comment, considering that this issue will occur for every loan with multiple lenders.

zrax-x

@Czar102 Yes, I agree with @0xDetermination. First of all, it is difficult for attackers to profit from it because attackers need to pay relatively expensive platform fees. Secondly, some lenders will indeed lose a certain amount of entry fee, but I think this loss is small (considering that the entry fee rate is 0.1%, and it is not a complete loss).

0xDetermination

@zrax-x Yes, but the entrance fee can be up to 10%, and even with a 0.1% fee rate a large amount of net loss can accumulate over time- which is why I think this issue could be borderline.

zrax-x

My opinion is that considering that the root cause of this issue and issue#41 are the same (both use the same distribution method, as @fann95 commented before), the difference is only in the fee calculation. At the same time, the impact of issue#41 is more serious (the attacker is profitable and the loss is greater). So, on both counts, I believe that its severity should be M.

Czar102

Because of the heavy constraints on the exploitability (is never profitable based on what was said), and the fact that (from my understanding) the goal of this fee is mainly to prevent extremely short-term borrows and not to increase lenders' earnings, I believe this is a Medium severity issue.

Planning to accept the escalation and downgrade the issue to Medium.

Czar102

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- qmdddd: accepted

0xDetermination

Fix looks good, fee collection has been reworked and entrance fees are added directly to the lender's reward balance.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: A borrower eligible for liquidation can pay an improperly large amount of fees, and may be unfairly liquidated

Source: https://github.com/sherlock-audit/2024-02-leverage-contracts-judging/issues/40

Found by

0xDetermination

Summary

If a borrower is partially liquidated and then increases the collateral balance to avoid further liquidation, they will pay an improperly large amount of fees and can be unfairly liquidated.

Vulnerability Detail

The root cause is that partial emergency liquidation doesn't update `accLoanRatePerSeconds` (https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L660-L666).

If a borrower is partially liquidated, fees will be increased by the entire collateral amount (https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L604-L633):

```
(collateralBalance, currentFees) = _calculateCollateralBalance(
    borrowing.borrowedAmount,
    borrowing.accLoanRatePerSeconds,
    borrowing.dailyRateCollateralBalance,
    accLoanRatePerSeconds
);
...
if (collateralBalance > 0) {
    ...
} else {
    currentFees = borrowing.dailyRateCollateralBalance; //entire collateral
 ↪  amount
}
...
borrowing.feesOwed += _pickUpPlatformFees(borrowing.holdToken, currentFees);
```

When liquidation occurs right after becoming liquidatable, the `collateralBalance` calculation in `repay()` above will be a small value like `-1`; and essentially all the fees owed will be collected.

If the borrower notices the partial liquidation and wishes to avoid further liquidation, `increaseCollateralBalance()` can be called to become solvent again. But since the `accLoanRatePerSeconds` wasn't updated, the borrower will have to doubly pay all the fees that were just collected. This will happen if a lender calls `harvest()` or the loan is liquidated again. The loan can also be liquidated unfairly, because the `collateralBalance` calculated above will be much lower than it should be.

Impact

The borrower may pay too many fees, and it's also possible to unfairly liquidate the position.

Code Snippet

https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L660-L666
https://github.com/sherlock-audit/2024-02-leverage-contracts/blob/main/wagmi-leverage/contracts/LiquidityBorrowingManager.sol#L604-L633

Tool used

Manual Review

Recommendation

Update `accLoanRatePerSeconds` for incomplete emergency liquidations.

Discussion

fann95

accLoanRatePerSeconds should not be updated since borrowedAmount are reduced, accordingly, a position debt also redused This issue was already discussed in the previous audit. Harvest cannot be called on a position that is under liquidation

nevillehuang

Invalid, agree with sponsors comments. This was previously discussed here

0xDetermination

Hi @nevillehuang @fann95 , posting commented PoC with console.logs below (modified test).

The flow is like so:

(a) Borrower's position is emergency liquidated by one lender

(b) Borrower doesn't want to get liquidated more so `increaseCollateralBalance()` is called

(c) Borrower will pay too many fees since `dailyRateCollateralBalance` was set to zero and `accLoanRatePerSeconds` wasn't updated

```
it("emergency repay will be successful for PosManNFT owner if the
↪  collateral is depleted", async () => {
    let debt: ILiquidityBorrowingManager.BorrowingInfoExtStructOutput[] =
        await borrowingManager.getBorrowerDebtsInfo(bob.address);
    console.log('collateralBalance', debt[0].collateralBalance);
    console.log('current collateral amount',
    ↪  debt[0].info.dailyRateCollateralBalance);
    console.log('estimated life time', debt[0].estimatedLifeTime);
    await time.increase(debt[0].estimatedLifeTime.toNumber() + 1);

    debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
    console.log('collateralBalance after advancing time',
    ↪  debt[0].collateralBalance);

    let borrowingKey = (await
    ↪  borrowingManager.getBorrowingKeysForBorrower(bob.address))[0];
    let deadline = (await time.latest()) + 60;

    let params: ILiquidityBorrowingManager.RepayParamsStruct = {
        returnOnlyHoldToken: true,
        isEmergency: true, //emergency
        internalSwapPoolfee: 0,
        externalSwap: [],
        borrowingKey: borrowingKey,
        minHoldTokenOut: BigNumber.from(0),
        minSaleTokenOut: BigNumber.from(0)
    };

    await expect(borrowingManager.connect(alice).repay(params, deadline))
        .to.emit(borrowingManager, "EmergencyLoanClosure")
        .withArgs(bob.address, alice.address, borrowingKey);

    debt = await borrowingManager.getBorrowerDebtsInfo(bob.address);
    console.log('collateralBalance after first liquidation (this is wrong,
        ↪  should be close to zero)', debt[0].collateralBalance); //this
        ↪  amount is wrong, way too large due to the borrower's collateral set
        ↪  to zero and accLoanRatePerSeconds not updated. We can see that the
        ↪  amount actually increased instead of decreasing as it should

    //borrower increases collateral by a large amount such that liquidation
        ↪  shouldn't be possible anymore, 18000000000000000000000 (this is
        ↪  currently scaled by collateral precision, 1e18)
    //This amount is about 75% of the orignal collateral amount of
        ↪  24948000000000000000000
    await borrowingManager.connect(bob).increaseCollateralBalance(borrowing⌋
        ↪  Key, 18000n, deadline); //adjust amount for collateral balance
        ↪  precision
    //below should revert since collateral balance was increased by a large
        ↪  amount, but the borrower gets liquidated
    await expect(borrowingManager.connect(bob).repay(params, deadline))
        .to.emit(borrowingManager, "EmergencyLoanClosure")
        .withArgs(bob.address, bob.address, borrowingKey);

    await expect(borrowingManager.connect(owner).repay(params, deadline))
        .to.emit(borrowingManager, "EmergencyLoanClosure")
        .withArgs(bob.address, owner.address, borrowingKey);
});
```

fann95

I see in the tests that the amount of debt has decreased by part of the closed Lp-share, but the rest of the debt has been maintained, as expected. collateralBalance after advancing time BigNumber { value: "-363497523148146947" } collateralBalance after first liquidation (this is wrong, should be close to zero) BigNumber { value: "-263884816316264683" } debt(263884816316264683) < debt(363497523148146947)

0xDetermination

@fann95 That's weird, when I run in the contest repo the output is: collateralBalance BigNumber { value: "623650275247685185305" } current collateral amount BigNumber { value: "24948000000000000000000" } estimated life time BigNumber { value: "21598" } collateralBalance after advancing time BigNumber { value: "-181748761574072227" } collateralBalance after first liquidation (this is wrong, should be close to zero) BigNumber { value: "-1811139262017361109770" }

Also, not sure if the test is passing on your end, but if it's passing I think the bug is there since the collateral is increased and emergency liq shouldn't happen

fann95

It looks like this is a valid issue..I got different results since I ran your test in the updated version. The PR which corrected the problems with the distribution of commissions also corrected this problem.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/RealWagmi/wagmi-leverage/commit/84416fcedfcc7eb062917bdc69f919bba9d3c0b7.

nevillehuang

@fann95 Is the root cause stemming from similar issues in #41 or only possible because of another issue, given fix PR is the same?

0xDetermination

@fann95 @nevillehuang Fix code looks good to me, PoC doesn't pass in the main repo.

In case my input is helpful- I think this issue is different than #41 since that issue describes a root cause/fix in `borrow()`, whereas the cause/fix for this issue is around not setting `dailyRateCollateralBalance` to zero in `repay()`. Additionally the PR is quite large and changes a lot of things.

nevillehuang

@fann95 @0xDetermination I am trying to figure out how the original fix could have fixed this issue without first considering it, which leads me to believe they share the same root causes revolving around distribution of fees. I would have to take a closer look at the fix PR.

fann95

> @fann95 @0xDetermination I am trying to figure out how the original fix could have fixed this issue without first considering it, which leads me to believe they share the same root causes revolving around distribution of fees. I would have to take a closer look at the fix PR.

This problem is indirectly related to the distribution of commissions. I got rid of the mechanism for accumulating a fee, therefore the current error was fixed.

nevillehuang

@Czar102 @0xDetermination What are your thoughts here based on duplication rules here? The core vulnerability seem to stem from erroneous distribution of fees which allowed for this issue to be possible in the first place. I am inclined to think this should be duplicated with #41

> 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.

0xDetermination

@nevillehuang @Czar102 The root cause and fix for this issue are both distinct from #41- the erroneous distribution of fees in #41 is caused by not harvesting fees when a new loan is taken with the `borrow()` function, and the fix is the new internal `_harvest()` function that runs in `borrow()`. (link)

This issue #40 is caused by the fee distribution mechanism in emergency liquidation mode in `repay()`, which is separate from `borrow()` and `harvest()`. The root cause and fix can both be seen here in `repay()`. The fix for #41 won't fix this issue.

nevillehuang

Since #39 and #41 talks about different types of fees, I agree they are not duplicates and they are both high severity findings. Given this erroneous fee calculation affects a large portion of the protocol, I agree with sponsor comments here and believe high severity is appropriate.

However, I believe that #40 is a medium severity issue only possible because of the root cause of wrong computation of fees for borrowed positions within #41. This is evident from the fix employed without considering this issue in the first place. Hence, I am going to duplicate it with #41 and assign it as high severity based on sherlock rules despite it having only a medium severity impact.

0xDetermination

Escalate

I understand @nevillehuang's point here, but I still think this shouldn't be a dup for the reasons I gave in my above comment. The fix PR changed a lot of things unrelated to validated issues, such as removal of the min fee mechanism.

Will appreciate @Czar102's decision.

Additionally, if validated as not a dup, not sure if this should be H or M based on the 'external conditions' criteria.

sherlock-admin2

> Escalate

> I understand @nevillehuang's point here, but I still think this shouldn't be a dup for the reasons I gave in my above comment. The fix PR changed a lot of things unrelated to validated issues, such as removal of the min fee mechanism.
>
> Will appreciate @Czar102's decision.
>
> Additionally, if validated as not a dup, not sure if this should be H or M based on the 'external conditions' criteria.

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.

Czar102

@nevillehuang can you elaborate on how does a fix to #40 follows from #41?

nevillehuang

@Czar102 I believe @fann95 answered it here and here

Based on sponsor description and comments above, The PR fix was performed without consideration of this issue, which led me to believe this issue only stems from the root cause of incorrect distribution.

Czar102

@nevillehuang what is the single logical error (maybe an assumption, maybe approach) that led to both of these issues?

Having issue A which makes the sponsor want to restructure code and it accidentally removing issue B doesn't make them duplicates.

fann95

I propose to confirm this issue and not consider it a duplicate.

nevillehuang

@Czar102 It stems from the logic for fee distribution. Although I disagree, seems like sponsor agrees to deduplicate so we can proceed with deduplication.

Czar102

@nevillehuang regarding this:

> It stems from the logic for fee distribution.

I believe it doesn't answer my question:

> what is the single logical error (maybe an assumption, maybe approach) that led to both of these issues?

I wanted to have a one-sentence description of the common ground of these issues, and the fact that the issues "stem from the logic for fee distribution" (are in the same part of the code logic) doesn't make them duplicates.

I'm planning to make this issue a unique issue, unless a justification (as mentioned above) is provided.

What are the considerations regarding the severity? @nevillehuang @0xDetermination what did you mean by the following fragment of your escalation?

> not sure if this should be H or M based on the 'external conditions' criteria

0xDetermination

@Czar102 I'm not 100% sure whether this is better suited for H or M, as I don't have a ton of experience with Sherlock judging rules. Basically, the issue can cause serious loss of funds (borrower's entire collateral), but it is conditional on a partial emergency liquidation followed by the borrower increasing collateral. It looks more like M to me but I don't want to speak too soon, will leave it up to you and @nevillehuang. Happy to provide more info if needed.

nevillehuang

Yup @Czar102 I agree with your decision, I cannot pinpoint an exact singular approach/code logic given this is an update contest and would take up too much time. The reason I duplicated them was on the side of caution, given the sponsor quite literally fix this issue without even considering it. But since sponsor also agree with deduplication, lets move ahead

Czar102

In that case, planning to make this issue a unique Medium severity one.

Czar102

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0xDetermination: accepted

0xDetermination

Fix looks good, collection of fees in emergency mode `repay()` has been reworked. Notably, `dailyRateCollateralBalance` is no longer set to zero.

sherlock-admin4

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.

SHERLOCK