



# Size Findings & Analysis Report

2024-09-16

## Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (4)
  - [H-01] When `sellCreditMarket()` is called to sell credit for a specific cash amount, the protocol might receive a lower swapping fee than expected
  - [H-02] Risk of overpayment due to race condition between `repay` and `liquidateWithReplacement` transactions
  - [H-03] The collateral remainder cap is incorrectly calculated during liquidation
  - [H-04] Users won't liquidate positions because the logic used to calculate the liquidator's profit is incorrect
- Medium Risk Findings (13)
  - [M-01] Multicall does not work as intended
  - [M-02] Users can not to buy/sell minimum credit allowed due to `exactAmountIn` condition
  - [M-03] Size uses wrong source to query available liquidity on Aave, resulting in borrow and lend operations being bricked upon mainnet deployment

- [M-04] Inadequate checks to confirm the correct status of the sequence/ sequencerUptimeFeed in PriceFeed.getPrice(). contract
- [M-05] Users may incur an unexpected fragmentation fee in the compensate(). call
- [M-06] Neither sellCreditMarket(). nor compensate(). checks whether the credit position to be sold is allowed for sale
- [M-07] Credit can be sold forcibly as forSale setting can be ignored via Compensate
- [M-08] Sandwich attack on loan fulfillment will temporarily prevent users from accessing their borrowed funds
- [M-09] Borrower is not able to compensate his lenders if he is underwater
- [M-10] withdraw(). users may can't withdraw underlyingBorrowToken properly
- [M-11] LiquidateWithReplacement does not charge swap fees on the borrower
- [M-12] executeBuyCreditMarket returns the wrong amount of cash and overestimates the amount that needs to be checked in the variable pool
- [M-13] Fragmentation fee is not taken if user compensates with newly created position
- Low Risk and Non-Critical Issues
  - L-01 There should be a grace period for repayment
  - L-02 Lenders can potentially lose funds in the buyCreditMarket function
  - L-03 Users who have a loan offer should have some USDC to create a loan .
  - L-04 The isMulticall flag is not correctly reset to false in the multicall function.
  - [L-05] Users cannot make compensation when the due date is within the minimum tenor period
  - [L-06] When the borrowAToken cap is reached, users may not always be able to repay their debt in some cases due to precision errors
- Disclosures

## Overview

---

## About C4

---

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Size smart contract system written in Solidity. The audit took place between June 10 — July 2, 2024.

## Wardens

---

108 Wardens contributed reports to Size:

1. [mt030d](#)
2. [KupiaSec](#)
3. [adeolu](#)
4. [elhaj](#)
5. [VAD37](#)
6. [pkqs90](#)
7. [serial-coder](#)
8. [ether\\_sky](#)
9. [hyh](#)
10. [Oxpiken](#)
11. [samuraii77](#)
12. [alix40](#)
13. [OxAlix2](#) ([a\\_kalout](#) and [ali\\_shehab](#))
14. [Oxhacksmithh](#)
15. [bin2chen](#)
16. [m4k2](#)
17. [stakog](#)
18. [AMOW](#)
19. [gesha17](#)
20. [Shield](#) ([Viraz](#), [OxA5DF](#), [Dravee](#) and [Udsen](#))
21. [grearlake](#)
22. [OxStalin](#)

23. ilchovski
24. BenRai
25. jsmi
26. almurhasan
27. Bob
28. dhank
29. zzebra83
30. trachev
31. inzinko
32. zarkk01
33. Honour
34. 3n0ch
35. said
36. zhaojohnson
37. silver\_eth
38. ayden
39. JCN
40. SpicyMeatball
41. Brenzee
42. OxRobocop
43. Nyx
44. DanielArmstrong
45. carlos\_alegre
46. OxJoyBoy03
47. asui
48. gd
49. Kalogerone
50. Infect3d
51. LinKenji
52. radin100
53. Jorgect

- 54. iam\_emptyset
- 55. BoltzmannBrain
- 56. Oxanmol
- 57. Oxarno
- 58. Ox04bytes
- 59. Sentryx (flacko and Buggsy)
- 60. shafLOW2
- 61. Bigsam
- 62. zanderbyte
- 63. Nihavent
- 64. OxRstStn
- 65. Rhaydden
- 66. ubl4nk
- 67. lanrebayode77
- 68. prapandey031
- 69. muellerberndt
- 70. hezze
- 71. MidgarAudits (vangrim and EVDoc)
- 72. Inspex (DeStinE21, ErbaZZ, Rugsurely, doggo, jokopoppo, mimic\_f and rxnnxchxi)
- 73. nnez
- 74. Oxrafaelnicolau
- 75. rscodes
- 76. evmboi32
- 77. nfmelendez
- 78. Zkillua
- 79. supersizerOx
- 80. OxBugSlayer
- 81. max10afternoon
- 82. iamandreiski
- 83. Nave765
- 84. Decipher100Eyes (shkim9323, mansub-song, rick and ssups)

- 85. ParthMandale
- 86. BajagaSec
- 87. TychaiOs
- 88. AlexCzm
- 89. K42
- 90. Bauchibred
- 91. atoko
- 92. NexusAudits (cheatcOd3 and Zanna)

This audit was judged by [hansfrieze](#).

Final report assembled by [thebrittfactor](#).

## Summary

---

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 11 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

## Scope

---

The code under review can be found within the [C4 Size repository](#), and is composed of 32 smart contracts written in the Solidity programming language and includes 2578 lines of Solidity code.

## Severity Criteria

---

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling

- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#), specifically our section on [Severity Categorization](#).

## 🔗 High Risk Findings (4)

---

### 🔗 [H-01] When `sellCreditMarket()`, is called to sell credit for a specific cash amount, the protocol might receive a lower swapping fee than expected

---

Submitted by [Oxpiken](#), also found by [3nOch](#) (1, 2), [OxAlix2](#), [Honour](#), [gd](#), [asui](#), [DanielArmstrong](#), [carlos\\_alegre](#), [zarkk01](#), [Shield](#) (1, 2), [mt030d](#), [Brenzee](#), [OxStalin](#), [dthank](#), [KupiaSec](#), [Kalogerone](#), [OxJoyBoy03](#), [bin2chen](#), [pkqs90](#), [ether\\_sky](#), [trachev](#), [samuraii77](#), [m4k2](#), and [LinKenji](#)

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/AccountingLibrary.sol#L249>

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/AccountingLibrary.sol#L256>

### 🔗 Impact

The protocol might receive a lower fee than expected when `sellCreditMarket()` is called to sell credit for a specific cash amount.

### 🔗 Proof of Concept

The protocol allows a user to sell their `credit` for `szaUSDC`, which can be used to redeem USDC thereafter.

- If a specific amount of `credit` is sold for `szaUSDC`, `sellCreditMarket()` will calculate the amount of `szaUSDC` received and fees paid to `feeRecipient`.
- If `credit` is sold for a specific amount of `szaUSDC`, `sellCreditMarket()` will calculate the amount of `credit` sold and fees paid to `feeRecipient`.

*Note: please see the calculation scenarios in warden's original submission.*

**Example 1:** Bob owned 120 `credit` and sell 120 `credit` to Candy for `szUSDC` . All results of the calculation are same as Example 1.

The sold credit and swap fee are calculated in `getCreditAmountIn().`:

Copy the below codes to [SellCreditMarketTest.t.sol](#) and run `forge test --match-test test_SellCreditMarket_sellCreditMarket_incorrectFee` :

8/89



```

        amount: 120e6,
        tenor: tenor,
        deadline: block.timestamp,
        maxAPR: apr,
        exactAmountIn: true
    })
);
//@audit-info alice has 0 szaUSDC left
assertEq(size.getUserView(alice).borrowATokenBalance, 0);
//@audit-info bob received 99.5e6 szaUSDC
assertEq(size.getUserView(bob).borrowATokenBalance,
99.5e6);
//@audit-info bob owed 120e6 debt
assertEq(size.getUserView(bob).debtBalance, 120e6);
//@audit-info feeRecipient received 0.5e6 szaUSDC as fee

assertEq(size.getUserView(feeRecipient).borrowATokenBalance,
500000);
//@audit-info restore to the snapshot before
sellCreditMarket
vm.revertTo(snapshot);
//@audit-info bob sell credit to alice for 99.5e6 szaUSDC
size.sellCreditMarket(
    SellCreditMarketParams({
        lender: alice,
        creditPositionId: RESERVED_ID,
        amount: 99.5e6,
        tenor: tenor,
        deadline: block.timestamp,
        maxAPR: apr,
        exactAmountIn: false
    })
);
//@audit-info alice has 2500 szaUSDC left
assertEq(size.getUserView(alice).borrowATokenBalance,
2500);
//@audit-info bob received 99.5e6 szaUSDC
assertEq(size.getUserView(bob).borrowATokenBalance,
99.5e6);
//@audit-info bob owed 120e6 debt
assertEq(size.getUserView(bob).debtBalance, 120e6);
//@audit-info feeRecipient received 497500 szaUSDC as fee

assertEq(size.getUserView(feeRecipient).borrowATokenBalance,
497500);
}

```

## Recommended Mitigation Steps

Correct the swap fee calculation:

```

function getCreditAmountIn(
    State storage state,
    uint256 cashAmountOut,
    uint256 maxCashAmountOut,
    uint256 maxCredit,
    uint256 ratePerTenor,
    uint256 tenor
) internal view returns (uint256 creditAmountIn, uint256 fees)
{
    uint256 swapFeePercent = getSwapFeePercent(state, tenor);

    uint256 maxCashAmountOutFragmentation = 0;

    if (maxCashAmountOut >= state.feeConfig.fragmentationFee)
    {
        maxCashAmountOutFragmentation = maxCashAmountOut -
state.feeConfig.fragmentationFee;
    }

    // slither-disable-next-line incorrect-equality
    if (cashAmountOut == maxCashAmountOut) {
        // no credit fractionalization

        creditAmountIn = maxCredit;
-       fees = Math.mulDivUp(cashAmountOut, swapFeePercent,
PERCENT);
+       fees = Math.mulDivUp(cashAmountOut, swapFeePercent,
PERCENT - swapFeePercent);
    } else if (cashAmountOut < maxCashAmountOutFragmentation)
    {
        // credit fractionalization

        creditAmountIn = Math.mulDivUp(
            cashAmountOut + state.feeConfig.fragmentationFee,
PERCENT + ratePerTenor, PERCENT - swapFeePercent
        );
-       fees = Math.mulDivUp(cashAmountOut, swapFeePercent,
PERCENT) + state.feeConfig.fragmentationFee;
+       fees = Math.mulDivUp(cashAmountOut +
state.feeConfig.fragmentationFee, swapFeePercent, PERCENT -
swapFeePercent) + state.feeConfig.fragmentationFee;
    } else {
        // for maxCashAmountOutFragmentation < amountOut <
maxCashAmountOut we are in an inconsistent situation
        // where charging the swap fee would require to sell
a credit that exceeds the max possible credit

        revert
Errors.NOT_ENOUGH_CASH(maxCashAmountOutFragmentation,
cashAmountOut);

```

```
}  
}
```

## Assessed type

Math

aviggiano (Size) confirmed and commented via duplicate Issue #213:

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/137>.*

hansfrieze (judge) increased severity to High

MotokoKusanagi-aka-Major (Size) commented:

*This issue is an incorrect implementation of the swap fees formula described in the technical documentation, resulting in less fee collection in the credit swapping process.*

*In our opinion, this should not be categorized as a “High” issue since it only negatively affects the protocol owners, because in the current version, there is no mechanism for fees redistribution, so no user category (lenders / credit buyers, borrowers / credit sellers, and liquidators) is negatively affected. In fact, the user who is subject to fee payment would have benefited from this issue since he would have ended up paying lower fees than expected.*

## [H-02] Risk of overpayment due to race condition between repay and liquidateWithReplacement transactions

Submitted by elhaj, also found by KupiaSec, mt030d, and VAD37

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/Size.sol#L198-L201>  
<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/Size.sol#L229-L244>

## Impact

- **Likelihood:** Medium - Users and liquidation bots are likely to call their respective functions around the same time.
- **Impact:** High - Users end up repaying more than double their future value.

## Proof of Concept

The `LiquidateWithReplacement` mechanism in the Size protocol allows a permissioned role ( `KEEPER_ROLE` ) to perform the liquidation of a debt position by replacing an unhealthy borrower with a healthier one from the order book. Here's how it works:

1. The debt position of the unhealthy borrower is liquidated, and the `futureValue` is paid to the Size contract by the liquidator, while the liquidator gets an equivalent amount plus reward in terms of collateral tokens from the unhealthy borrower.
2. The borrower of this `debtPosition` is updated to the new healthier borrower.
3. The debt position retains its same `futureValue` and `ID`.
4. A portion of the repaid funds is used to fund a new borrower.

```
function executeLiquidateWithReplacement(State storage state,
LiquidateWithReplacementParams calldata params)
    external
    returns (uint256 issuanceValue, uint256
liquidatorProfitCollateralToken, uint256
liquidatorProfitBorrowToken)
{
    // some code ...
/*1*/>>    liquidatorProfitCollateralToken =
state.executeLiquidate(LiquidateParams({debtPositionId:
params.debtPositionId, minimumCollateralProfit:
params.minimumCollateralProfit}));

    uint256 ratePerTenor = borrowOffer.getRatePerTenor(
        VariablePoolBorrowRateParams({
            variablePoolBorrowRate:
state.oracle.variablePoolBorrowRate,
            variablePoolBorrowRateUpdatedAt:
state.oracle.variablePoolBorrowRateUpdatedAt,
            variablePoolBorrowRateStaleRateInterval:
state.oracle.variablePoolBorrowRateStaleRateInterval
        }),
        tenor
    );
```

```

        issuanceValue =
Math.mulDivDown(debtPositionCopy.futureValue, PERCENT, PERCENT +
ratePerTenor);
        liquidatorProfitBorrowToken = debtPositionCopy.futureValue
- issuanceValue;

/*2*/>>    debtPosition.borrower = params.borrower;
/*3*/>>    debtPosition.futureValue =
debtPositionCopy.futureValue;
        debtPosition.liquidityIndexAtRepayment = 0;

        emit Events.UpdateDebtPosition(params.debtPositionId,
debtPosition.borrower, debtPosition.futureValue,
debtPosition.liquidityIndexAtRepayment);
        // @note : in this case the borrower doesn't pay swapFees
        state.data.debtToken.mint(params.borrower,
debtPosition.futureValue);
/*4*/>>    state.data.borrowAToken.transferFrom(address(this),
params.borrower, issuanceValue);
        state.data.borrowAToken.transferFrom(address(this),
state.feeConfig.feeRecipient, liquidatorProfitBorrowToken);
    }

```

By maintaining the same debt position ID and future value, lenders (all credit holders of this debt position) can still expect repayment on the original due date.

On the other hand, a user can always `repay` a debt position. To repay, the user specifies the ID of the debt position and calls the `repay` function, which repays the loan of the specified debt position.

```

function executeRepay(State storage state, RepayParams
calldata params) external {
    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);

    state.data.borrowAToken.transferFrom(msg.sender,
address(this), debtPosition.futureValue);
    debtPosition.liquidityIndexAtRepayment =
state.data.borrowAToken.liquidityIndex();
    state.repayDebt(params.debtPositionId,
debtPosition.futureValue);

    emit Events.Repay(params.debtPositionId);
}

```

The issue arises when a borrower sends a transaction to `repay` their debt position when it becomes liquidatable, but a `liquidateWithReplacement` transaction is executed just before the user's transaction. This way, the user will be liquidated, and their collateral will be used to

repay debt. Since `liquidateWithReplacement` only changes the borrower of a `debtPosition`, the user ends up paying the debt of another borrower immediately, resulting in the user repaying more than double their future value (more than 2x because of liquidation rewards).

This doesn't require a malicious keeper (who calls `liquidateWithReplacement`) as it's trusted, it can happen unintentionally, and there is a pretty good chance for this to occur; as users and the liquidator (keeper bot in this case) will probably call their respective functions around the same time.

When a user notices that their position is liquidatable (probably through a UI notification), there is a high chance they will repay their debt to avoid liquidation, and the same goes for the liquidation bot. Thus, there is a good chance that the `liquidateWithReplacement` transaction will be executed before the user's repayment transaction. The impact is high here, as the user ends up paying 2x their debt.

In case `liquidateWithReplacement` becomes permissionless and anyone can call it, this will be critically exploitable. An exploit scenario would be:

1. The collateral token price drops, making Bob's position liquidatable.
2. Bob gets notified and sends a transaction to repay his debt position with `ID=0`.
3. Alice sees Bob's transaction in the mempool and front-runs him to liquidate the debt position with `ID=0`, setting herself as the borrower (it can set any curve borrow offer). An equivalent amount to the `futureValue` in terms of collateral tokens is sent from Bob's account to Alice, and Alice receives another cash of `borrowAToken` since she's the borrower.
4. Bob's transaction gets executed after that, and Bob ends up paying Alice's debt. Bob suffers loss.

## Recommended Mitigation Steps

To prevent unintended debt repayment due to borrower changes, users should specify both the borrower address and the debt position ID when repaying. This ensures users have control over who they are repaying for, avoiding scenarios where they might repay another borrower's debt due to a recent `liquidateWithReplacement` transaction.

Make this changes in [repay lib](#):

```
struct RepayParams {
    uint256 debtPositionId;
+   address borrower;
}
```

```

function validateRepay(State storage state, RepayParams
calldata params) external view {
    // validate debtPositionId
    if (state.getLoanStatus(params.debtPositionId) ==
LoanStatus.REPAID) {
        revert
Errors.LOAN_ALREADY_REPAID(params.debtPositionId);
    }
+    if (state.getDebtPosition(params.debtPositionId).borrower
!= params.borrower) revert("invalid borrower");

    // validate msg.sender
    // N/A
}

```

## Assessed type

### Timing

#### aviggiano (Size) acknowledged and commented:

*Feedback from the team:*

*This looks valid since it does not look like a user mistake, it is a kind of MEV. In practice, I think it can happen rarely since for this to happen we need this to be true:*

*When a user notices that their position is liquidatable (probably through a UI notification), there is a high chance they will repay their debt to avoid liquidation,*

*In my honest opinion, there is a high chance they deposit more collateral or compensate to avoid liquidation, since repaying early would mean losing all the interest; so not a great way to do that (even though in some cases it could be the only chance). Technically valid, but practically low impact.*

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/141>.*

#### hansfrieze (judge) commented:

*I will keep as High because the impact is critical and the likelihood is not rare.*

**OxStalin (warden) commented:**

*This is an interesting case, but, a high severity seems to be overgenerous because of the pre-conditions that needs to exist for this to become a problem.*

1. There needs to be a borrower for the APR and Tenor of the liquidated Position.
2. The borrower being liquidated needs to attempt to repay the same position being liquidated (Borrower can have multiple debt positions and can attempt to repay any of the others, not necessarily the exact position being liquidated and replaced).
3. The liquidate and replacement transaction must be executed BEFORE than the repay tx.

*@hansfrieese - because of the multiple on-chain conditions, this seems to fit more of a medium severity.*

**hansfrieese (judge) commented:**

*I agree that it requires some prerequisites. However, with a liquidatable position, it is not that strong assumption that those 2 functions could be called simultaneously. According to the C4 severity categorization, High is more appropriate due to the direct fund loss and the likelihood of this scenario occurring.*

**MotokoKusanagi-aka-Major (Size) commented:**

*This issue arises from a potential race condition involving two function calls regarding a debt position eligible for liquidation:*

- A borrower calling `repay()` to avoid the liquidation and
- The protocol backend calling `liquidateWithReplacement()` to liquidate the position with replacement (this is a permissioned method).



Since there was no check regarding the ownership of that debt, any address was allowed to repay the debt of any other address so if the `liquidateWithReplacement()` was executed first, the borrower would have ended up being liquidated and repaying someone else debt, because of the replacement.

The impact is certainly high but the likelihood of this issue happening in our opinion has been largely overestimated, and is for all practical purposes, non-existent.

The reason is that a borrower eligible for liquidation has no incentive to repay loans to avoid liquidation since he would forfeit the interest for the remaining lifetime of the loan. In that case, what makes more sense is for them to increase their collateral ratio by

1. Depositing more collateral and
2. Compensating some of their debt with eligible credit

The race between 1 and 2 with `liquidateWithReplacement()` is not associated with any issue.

The only case where the borrower can make sense to repay his loan early is when the forfeited interests would be negligible, meaning at the end of the loan lifecycle. However, in that case, the replacement is extremely unlikely to happen since it would require a borrower willing to borrow passively for a very short tenor. This is very unlikely to happen as a borrower would need some time to react to his limit borrow order being filled and quickly decide what to do with the borrowed money, before this new loan goes overdue and he becomes eligible for liquidation.

On top of this, the `liquidateWithReplacement()` method is permissioned and we have a negligible economic incentive in running it vs a standard liquidation. This is because the latter is unaffected by the issue mentioned above since the value the protocol captures from running a liquidation with replacement is proportional to the remaining loan lifetime. It would therefore be a net loss to use `liquidateWithReplacement()` instead of `liquidate()` for a loan very close to maturity as we would gain almost nothing while potentially significantly affecting our users.

## [H-03] The collateral remainder cap is incorrectly calculated during liquidation

Submitted by [ether\\_sky](#), also found by [hyh](#), [samurair77](#), [m4k2](#), [KupiaSec](#), [stakog](#), [AMOW](#), [jsmi](#), and [Bob](#)

When a position is overdue, it can be liquidated even if the owner has a sufficient collateral ratio. Some rewards are sent to the liquidator as an incentive and some fees are assigned to the protocol. However, the calculation of the protocol fee is flawed due to an incorrect collateral remainder cap calculation.

Imagine the collateral ratio for liquidation is set at 130%. The collateral equivalent to the debt should be paid to the protocol, and the excess collateral (30%) can be the remainder cap. Currently, the entire 130% is treated as the remainder cap, leading to an unfair situation where a user with a 150% collateral ratio experiences more loss than a user with a 140% CR.

This discourages users with higher CRs, who generally contribute to the protocol's health. If higher CRs result in greater losses, it disincentivizes users from providing sufficient collateral to the protocol.

## Proof of Concept

Imagine Bob and Candy have the same debt position, and these positions are overdue.

- The collateral ratio for liquidation is 130%.
- Candy has a 150% collateral ratio, and Bob has a 140% CR.

Both positions are liquidated by liquidators:

```
function executeLiquidate(State storage state, LiquidateParams
calldata params)
    external
    returns (uint256 liquidatorProfitCollateralToken)
{
    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);
    LoanStatus loanStatus =
state.getLoanStatus(params.debtPositionId);
    uint256 collateralRatio =
state.collateralRatio(debtPosition.borrower)

    uint256 collateralProtocolPercent =
state.isUserUnderwater(debtPosition.borrower)
        ? state.feeConfig.collateralProtocolPercent
        : state.feeConfig.overdueCollateralProtocolPercent;

    uint256 assignedCollateral =
state.getDebtPositionAssignedCollateral(debtPosition); // @1
```

```

uint256 debtInCollateralToken =
state.debtTokenAmountToCollateralTokenAmount(debtPosition.futureVa
lue); // @2
uint256 protocolProfitCollateralToken = 0;

// profitable liquidation
if (assignedCollateral > debtInCollateralToken) {
    uint256 liquidatorReward = Math.min(
        assignedCollateral - debtInCollateralToken,
        Math.mulDivUp(debtPosition.futureValue,
state.feeConfig.liquidationRewardPercent, PERCENT)
    ); // @3
    liquidatorProfitCollateralToken = debtInCollateralToken +
liquidatorReward;

    // split the remaining collateral between the protocol and
the borrower, capped by the crLiquidation
    uint256 collateralRemainder = assignedCollateral -
liquidatorProfitCollateralToken; // @4

    // cap the collateral remainder to the liquidation
collateral ratio
    // otherwise, the split for non-underwater overdue loans
could be too much
    uint256 collateralRemainderCap =
        Math.mulDivDown(debtInCollateralToken,
state.riskConfig.crLiquidation, PERCENT); // @5

    collateralRemainder = Math.min(collateralRemainder,
collateralRemainderCap);

    protocolProfitCollateralToken =
Math.mulDivDown(collateralRemainder, collateralProtocolPercent,
PERCENT); // @6
}
}

```

@1 : Calculate assigned collaterals to the debt position. The amount assigned from Candy's collateral is larger than that from Bob's.

@2 : Convert the debt token to the collateral token. The converted amounts are the same for both users.

@3 : Calculate the reward for liquidation. These amounts are the same because the debt sizes are the same and both users have enough CR to cover the profit.

@4 : Calculate the collateral remainder. Candy obviously has a larger amount.

@5 : Calculate the collateral remainder cap. The cap should be 30%, but it is currently calculated as 130%, which is the CR for liquidation itself.

@6 : As a result, Candy will experience more loss than Bob due to their excess amounts (50%, 40%) being below the cap (130%).

If users know that they will experience more loss when they have a larger CR, they will tend to maintain their CR as low as possible. While they will try to avoid liquidation, there will be no users with a CR higher than, for example, 200%. This is detrimental to the protocol.

Please check below log:

```

future value of Bob    => 600000000
future value of Candy => 600000000
assigned collateral of Bob    => 1000000000000000000000
assigned collateral of Candy => 1200000000000000000000
debt in collateral token of Bob    => 600000000000000000000
debt in collateral token of Candy => 600000000000000000000
liquidator reward of Bob    => 3000000
liquidator reward of Candy => 3000000
liquidator profit of collateral token of Bob    =>
60000000000000003000000
liquidator profit of collateral token of Candy =>
60000000000000003000000
collateral remainder of Bob    => 3999999999997000000
collateral remainder of Candy => 5999999999997000000
collateral remainder cap of Bob    => 780000000000000000000
collateral remainder cap of Candy => 780000000000000000000
collateral remainder of Bob    => 3999999999997000000
collateral remainder of Candy => 5999999999997000000
protocol profit collateral token of Bob    =>
39999999999970000
protocol profit collateral token of Candy =>
59999999999970000

```

Please add below test to the `test/local/actions/Liquidate.t.sol` : (use `--var-ir` flag):

## ► Details

## 🔗 Recommended Mitigation Steps

```

function executeLiquidate(State storage state, LiquidateParams
calldata params)
    external
    returns (uint256 liquidatorProfitCollateralToken)
{
    if (assignedCollateral > debtInCollateralToken) {
        uint256 liquidatorReward = Math.min(
            assignedCollateral - debtInCollateralToken,
            Math.mulDivUp(debtPosition.futureValue,
state.feeConfig.liquidationRewardPercent, PERCENT)
        );
        liquidatorProfitCollateralToken =
debtInCollateralToken + liquidatorReward;
    }
}

```

```

uint256 collateralRemainder = assignedCollateral -
liquidatorProfitCollateralToken;

-      uint256 collateralRemainderCap =
        Math.mulDivDown(debtInCollateralToken,
state.riskConfig.crLiquidation, PERCENT);
+      uint256 collateralRemainderCap =
        Math.mulDivDown(debtInCollateralToken,
state.riskConfig.crLiquidation - PERCENT, PERCENT);
    }
}

```

## Assessed type

Math

aviggiano (Size) confirmed via duplicate Issue #140

hansfrieze (judge) increased severity to High and commented:

*Marking this report as primary with the detailed POC.*

MotokoKusanagi-aka-Major (Size) commented:

*We acknowledge the severity of these issues being high since they impact the users negatively and are not unlikely to happen, but we want to share a bit more context of how these bugs have been introduced into the code*

*H-03 and H-04 are two incorrect implementations of formulas correctly specified in the tech documentation, much like H-01. Note that this means that three of the four high-severity issues were simply incorrect implementations of formulas.*

*Incorrect implementations of this sort are usually easily caught by simple unit tests relying on pre-computed expected values so it is uncommon to find them in audit reports—and indeed numerous wardens also caught these bugs during the C4 competition—so we want to explain why this happened to avoid incorrect conclusions about our development process.*

*One finding of the Spearbit Audit (the one before the C4) required us to redesign and reimplement the fee mechanism entirely.*

*The changes required were quite significant and we had a tight deadline since we had already defined C4 start date and the launch date. This required us to be very judicious with our time and focus on the actual implementation and be light on tests to avoid having to reschedule the C4 competition.*

*However, since what ultimately matters the most for us is the security of our code, we worked on the tests in parallel with the competition and we were able to identify these bugs even before they were also discovered by the C4 competitors.*

*In conclusion, we were aware the strategy we had chosen would have increased the risk of having more bugs found during the competition—we did not optimize for receiving “high grades” from the competition (even though overall the “grades” we received seem pretty much aligned with the ones of other protocols). We chose the strategy to be able to meet the deadlines for the competition, and ultimately it did not come at the cost of less code security.*

See also [Issue #21](#).

## 🔗 [H-04] Users won't liquidate positions because the logic used to calculate the liquidator's profit is incorrect

Submitted by [ether\\_sky](#), also found by [serial-coder](#), [samurai77](#), [muellerberndt](#), [OxAlix2](#), [3n0ch](#), [nfmelendez](#), [BenRai](#), [Zkillua](#), [supersizerOx](#), [ayden](#), [gd](#), [OxBugSlayer](#), [radin100](#), [DanielArmstrong](#), [max10afternoon](#), [Bigsam](#), [iamandreiski](#), [asui](#), [Oxanmol](#), [Honour](#), [OxStalin](#), [Jorgect](#), [Nave765](#), [zanderbyte](#), [carlos\\_alegre](#), [Nyx](#), [Decipher100Eyes](#), [inzinko \(1, 2\)](#), [dthank](#), [Infect3d](#), [adeolu](#), [ParthMandale](#), [Oxarno](#), [Oxpiken](#), [hezze](#), [elhaj](#), [trachev](#), [BajagaSec](#), [silver\\_eth](#), [Nihavent](#), [MidgarAudits](#), [zarkk01](#), [hyh](#), [TychaiOs](#), [alix40](#), [OxRstStn](#), [Brenzee](#), [KupiaSec](#), [Kalogerone](#), [OxJoyBoy03](#), [ilchovski](#), [OxRobocop](#), [stakog](#), [bin2chen](#), [AlexCzm](#), [AMOW](#), [pkqs90](#), [VAD37](#), [Sentryx](#), and [said](#)

Liquidation is crucial, and all liquidatable positions must be liquidated to maintain the protocol's solvency. However, because the liquidator's profit is calculated incorrectly, users are not incentivized to liquidate unhealthy positions. This issue has a significant impact on the protocol.

## 🔗 Proof of Concept

When the collateral ratio of a user falls below the specified threshold, the user's debt positions become liquidatable.

The liquidation process involves the following steps:

```
function executeLiquidate(State storage state, LiquidateParams
calldata params)
    external
    returns (uint256 liquidatorProfitCollateralToken)
{
    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);
    LoanStatus loanStatus =
state.getLoanStatus(params.debtPositionId);
    uint256 collateralRatio =
state.collateralRatio(debtPosition.borrower);

    uint256 collateralProtocolPercent =
state.isUserUnderwater(debtPosition.borrower)
    ? state.feeConfig.collateralProtocolPercent
    : state.feeConfig.overdueCollateralProtocolPercent;

    uint256 assignedCollateral =
state.getDebtPositionAssignedCollateral(debtPosition); // @audit,
step 1
    uint256 debtInCollateralToken =
state.debtTokenAmountToCollateralTokenAmount(debtPosition.futureVa
lue); // @audit, step 2
    uint256 protocolProfitCollateralToken = 0;

    if (assignedCollateral > debtInCollateralToken) {
        uint256 liquidatorReward = Math.min(
            assignedCollateral - debtInCollateralToken,
            Math.mulDivUp(debtPosition.futureValue,
state.feeConfig.liquidationRewardPercent, PERCENT)
        ); // @audit, step 3
        liquidatorProfitCollateralToken =
debtInCollateralToken + liquidatorReward;

        ...
    } else {
        liquidatorProfitCollateralToken = assignedCollateral;
    }
}
```

**Step 1:** Calculate the collateral assigned to the debt.

- For example, if the collateral is 100 ETH and the total debt is 1000 USDC, and the liquidated position's debt is 200 USDC, then 20 ETH is assigned to this debt.

**Step 2:** Convert debt tokens to collateral tokens.

- For instance, if 1 ETH equals 1000 USDC and the liquidated position's debt is 200 USDC, then this value converts to 0.2 ETH.

**Step 3:** Calculate the liquidator's profit if the assigned collateral to this position is larger than the debt value.

- It's crucial to note that `assignedCollateral` and `debtInCollateralToken` values are denominated in ETH's decimal (18), whereas `debtPosition.futureValue` is denominated in USDC's decimal (6).
- As a result, the second value becomes extremely small, which fails to fully incentivize liquidators. This discrepancy is because the calculated profit is equal to the exact profit divided by `1e12`.

As you can see in the log below, the profit is calculated to be extremely small:

```
future value in usdc      ==> 82814071
future value in WETH      ==> 82814071000000000000
assigned collateral in WETH ==> 100000000000000000000
*****
liquidator profit in WETH ==> 4140704
exact profit in WETH      ==> 41407035500000000000
```

Please add below test to the `test/local/actions/Liquidate.t.sol`:

```
function test_liquidate_profit_of_liquidator() public {
    uint256 rewardPercent =
size.feeConfig().liquidationRewardPercent;

    // current reward percent is 5%
    assertEq(rewardPercent, 50000000000000000);

    _deposit(alice, weth, 100e18);
    _deposit(alice, usdc, 100e6);
    _deposit(bob, weth, 100e18);
    _deposit(liquidator, weth, 100e18);
    _deposit(liquidator, usdc, 100e6);

    _buyCreditLimit(alice, block.timestamp + 365 days,
YieldCurveHelper.pointCurve(365 days, 0.03e18));
    uint256 debtPositionId = _sellCreditMarket(bob, alice,
RESERVED_ID, 80e6, 365 days, false);

    _setPrice(1e18);
```



```

assertTrue(size.isDebtPositionLiquidatable(debtPositionId));

    uint256 futureValue =
size.getDebtPosition(debtPositionId).futureValue;
    uint256 assignedCollateral =
size.getDebtPositionAssignedCollateral(debtPositionId);
    uint256 debtInCollateralToken =

size.debtTokenAmountToCollateralTokenAmount(futureValue);

    console2.log('future value in usdc      ==> ',
futureValue);
    console2.log('future value in WETH      ==> ',
debtInCollateralToken);
    console2.log('assigned collateral in WETH ==> ',
assignedCollateral);

    console2.log('*****');

    uint256 liquidatorProfit = _liquidate(liquidator,
debtPositionId, 0);
    console2.log('liquidator profit in WETH ==> ',
liquidatorProfit - debtInCollateralToken);

    uint256 exactProfit = Math.min(
        assignedCollateral - debtInCollateralToken,
        Math.mulDivUp(debtInCollateralToken, rewardPercent,
PERCENT)
    );
    console2.log('exact profit in WETH      ==> ',
exactProfit);
}

```

## Recommended Mitigation Steps

Use the converted value to ETH :

```

function executeLiquidate(State storage state, LiquidateParams
calldata params)
    external
    returns (uint256 liquidatorProfitCollateralToken)
{
    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);
    LoanStatus loanStatus =
state.getLoanStatus(params.debtPositionId);
    uint256 collateralRatio =
state.collateralRatio(debtPosition.borrower);

```

```

uint256 collateralProtocolPercent =
state.isUserUnderwater(debtPosition.borrower)
    ? state.feeConfig.collateralProtocolPercent
    : state.feeConfig.overdueCollateralProtocolPercent;

uint256 assignedCollateral =
state.getDebtPositionAssignedCollateral(debtPosition);
uint256 debtInCollateralToken =
state.debtTokenAmountToCollateralTokenAmount(debtPosition.futureValue);
uint256 protocolProfitCollateralToken = 0;

if (assignedCollateral > debtInCollateralToken) {
    uint256 liquidatorReward = Math.min(
        assignedCollateral - debtInCollateralToken,
        -
        Math.mulDivUp(debtPosition.futureValue,
state.feeConfig.liquidationRewardPercent, PERCENT)
        +
        Math.mulDivUp(debtInCollateralToken,
state.feeConfig.liquidationRewardPercent, PERCENT)
    );
    liquidatorProfitCollateralToken =
debtInCollateralToken + liquidatorReward;

    ...
} else {
    liquidatorProfitCollateralToken = assignedCollateral;
}
}

```

## Assessed type

Math

aviggiano (Size) confirmed and commented:

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/118>*

MotokoKusanagi-aka-Major (Size) commented:

*We acknowledge the severity of these issues being high since they impact the users negatively and are not unlikely to happen, but we want to share a bit more context of how these bugs have been introduced into the code*

*H-03 and H-04 are two incorrect implementations of formulas correctly specified in the tech documentation, much like H-01. Note that this means that three of the four high-severity issues were simply incorrect implementations of formulas.*

*Incorrect implementations of this sort are usually easily caught by simple unit tests relying on pre-computed expected values so it is uncommon to find them in audit reports—and indeed numerous wardens also caught these bugs during the C4 competition—so we want to explain why this happened to avoid incorrect conclusions about our development process.*

*One finding of the Spearbit Audit (the one before the C4) required us to redesign and reimplement the fee mechanism entirely.*

*The changes required were quite significant and we had a tight deadline since we had already defined C4 start date and the launch date. This required us to be very judicious with our time and focus on the actual implementation and be light on tests to avoid having to reschedule the C4 competition.*

*However, since what ultimately matters the most for us is the security of our code, we worked on the tests in parallel with the competition and we were able to identify these bugs even before they were also discovered by the C4 competitors.*

*In conclusion, we were aware the strategy we had chosen would have increased the risk of having more bugs found during the competition—we did not optimize for receiving “high grades” from the competition (even though overall the “grades” we received seem pretty much aligned with the ones of other protocols). We chose the strategy to be able to meet the deadlines for the competition, and ultimately it did not come at the cost of less code security.*

*See also [Issue #70](#).*

## 🔗 Medium Risk Findings (13)

---

### 🔗 [M-01] Multicall does not work as intended

---

*Submitted by [prapandey031](#), also found by [Inspex](#), [samuraii77](#), [muellerberndt](#), [3n0ch](#), [Jorgect](#), [Honour](#), [nnez](#), [MidgarAudits](#), [elhaj](#), [alix40](#), [OxRobocop](#), [Oxrafaelnicolau](#), [mt030d](#), [inzinko](#), [rscodes](#), [Oxpiken](#), [hezze](#), [evmboi32](#), [zarkk01](#), [trachev](#), [KupiaSec](#), [stakog](#),*

[BoltzmannBrain](#), [bin2chen](#), [pkqs90](#), [shaflow2](#), [VAD37](#), [Sentryx](#), [ether\\_sky](#), [said](#), and [SpicyMeatball](#)

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/Multicall.sol#L29>  
<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/Multicall.sol#L37>

## Impact

The `multicall(bytes[] calldata _data)` function in the `Size.sol` contract does not work as intended. The intention of the `multicall(bytes[] calldata _data)` function is to allow users to access multiple functionalities of the `Size` protocol, such as a (deposit and repay) pair, by a single transaction to `Size.sol`:

<https://github.com/code-423n4/2024-06-size/blob/main/src/Size.sol#L142>

```
function multicall(bytes[] calldata _data)
    public
    payable
    override(IMulticall)
    whenNotPaused
    returns (bytes[] memory results)
{
    results = state.multicall(_data);
}
```

The `multicall` function allows batch processing of multiple interactions with the protocol in a single transaction. This also allows users to take actions that would otherwise be denied due to deposit limits. One of these actions is a (deposit and repay) pair.

Let's say a credit-debt pair exists. Assume that the tenor of the debt is 1 year and the future value is `100ke6 USDC`. Let's say the borrower decides to repay the loan just 1 day before the maturity ends. During this 1 year, the total supply of `borrowAToken` had increased so much that the total supply of `borrowAToken` was just `10e6 USDC` worth below the cap (that is, just below the cap) at the time when the borrower decided to repay the loan.

To repay a loan, `Size` requires the user to have sufficient `borrowAToken` :

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Repay.sol#L49>

```
function executeRepay(State storage state, RepayParams
calldata params) external {
    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);
```

```

        state.data.borrowAToken.transferFrom(msg.sender,
address(this), debtPosition.futureValue);
        debtPosition.liquidityIndexAtRepayment =
state.data.borrowAToken.liquidityIndex();
        state.repayDebt(params.debtPositionId,
debtPosition.futureValue);

        emit Events.Repay(params.debtPositionId);
    }
}

```

The user achieves this by first depositing the required amount of underlying borrow tokens (here, USDC) and then calling the `repay(RepayParams calldata params)` function:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/DepositTokenLibrary.sol#L49>

```

function depositUnderlyingBorrowTokenToVariablePool(State
storage state, address from, address to, uint256 amount)
external
{
    state.data.underlyingBorrowToken.safeTransferFrom(from,
address(this), amount);

    IAToken aToken =

    IAToken(state.data.variablePool.getReserveData(address(state.data.
underlyingBorrowToken)).aTokenAddress);

    uint256 scaledBalanceBefore =
aToken.scaledBalanceOf(address(this));

    state.data.underlyingBorrowToken.forceApprove(address(state.data.v
ariablePool), amount);

    state.data.variablePool.supply(address(state.data.underlyingBorrow
Token), amount, address(this), 0);

    uint256 scaledAmount =
aToken.scaledBalanceOf(address(this)) - scaledBalanceBefore;

    state.data.borrowAToken.mintScaled(to, scaledAmount);
}

```

Now, in our case, when the borrower decides to deposit `100ke6` USDC (at max if it would have some existing `borrowAToken`), he would not be able to do so (as the cap would be hit by depositing just `10e6` USDC). The situation is that the tenor is about to end (1 day left) and the

borrower is not able to repay, not because he does not have money, but because `borrowAToken` 's total supply cap does not allow him to deposit enough USDC.

To mitigate this, Size provides the multicall function, that bypasses the deposit limit and allows users to carry out such actions (LOC-80 in `Deposit.sol`):

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Deposit.sol#L80>

```
function executeDeposit(State storage state, DepositParams
calldata params) public {
    address from = msg.sender;
    uint256 amount = params.amount;
    if (msg.value > 0) {
        // do not trust msg.value (see `Multicall.sol`)
        amount = address(this).balance;
        // slither-disable-next-line arbitrary-send-eth
        state.data.weth.deposit{value: amount}();
        state.data.weth.forceApprove(address(this), amount);
        from = address(this);
    }

    if (params.token ==
address(state.data.underlyingBorrowToken)) {
        state.depositUnderlyingBorrowTokenToVariablePool(from,
params.to, amount);
        // borrow aToken cap is not validated in multicall,
        // since users must be able to deposit more tokens
to repay debt
        if (!state.data.isMulticall) {
            state.validateBorrowATokenCap();
        }
    } else {
        state.depositUnderlyingCollateralToken(from,
params.to, amount);
    }

    emit Events.Deposit(params.token, params.to, amount);
}
}
```

Let's say a user uses the multicall function for a (deposit and repay) pair action. The multicall function checks for an invariant that restricts users from depositing more `borrowATokens` than required to repay the loan by restricting:

increase in `borrowAToken` supply <= decrease in `debtToken` supply

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L19>

```
function validateBorrowATokenIncreaseLteDebtTokenDecrease(
    State storage state,
    uint256 borrowATokenSupplyBefore,
    uint256 debtTokenSupplyBefore,
    uint256 borrowATokenSupplyAfter,
    uint256 debtTokenSupplyAfter
) external view {
    // If the supply is above the cap
    if (borrowATokenSupplyAfter >
state.riskConfig.borrowATokenCap) {
        uint256 borrowATokenSupplyIncrease =
borrowATokenSupplyAfter - borrowATokenSupplyBefore
        ? borrowATokenSupplyAfter -
borrowATokenSupplyBefore
        : 0;
        uint256 debtATokenSupplyDecrease =
debtTokenSupplyBefore - debtTokenSupplyAfter ?
debtTokenSupplyBefore - debtTokenSupplyAfter : 0;

        // and the supply increase is greater than the debt
reduction
        if (borrowATokenSupplyIncrease >
debtATokenSupplyDecrease) {
            // revert
            revert
Errors.BORROW_ATOKEN_INCREASE_EXCEEDS_DEBT_TOKEN_DECREASE(
                borrowATokenSupplyIncrease,
debtATokenSupplyDecrease
            );
        }
        // otherwise, it means the debt reduction was greater
than the inflow of cash: do not revert
    }
    // otherwise, the supply is below the cap: do not revert
}
```

The problem is, this is the exact invariant that is broken. The PoC below explains how in detail.

## Proof of Concept

I have provided a thorough step-by-step PoC to explain how the invariant is broken. Let's continue with our previous example.

Let's say the `borrowAToken` cap is `1Me6` and its current supply is `990ke6`. Before calling the `multicall` function, the `borrowAToken` balances are:

- `borrower = 0`

- Size contract = 0
- Total supply of borrowAToken = 990ke6

Also, before calling the multicall function:

```
debtToken.balanceOf[borrower] = 100ke6
```

The borrower calls the multicall function, with (deposit and repay) action pair in Size.sol:

<https://github.com/code-423n4/2024-06-size/blob/main/src/Size.sol#L142>

```
function multicall(bytes[] calldata _data)
    public
    payable
    override(IMulticall)
    whenNotPaused
    returns (bytes[] memory results)
{
    results = state.multicall(_data);
}
```

To make things simple to understand, let's assume that this is the first repayment of a loan; that is, other credit-debt pairs exist but have sufficient time to mature.

The borrower has used the multicall to deposit 500ke6 USDC and would want to repay his debt completely (100ke6 USDC). The deposit action would be performed (as the deposit limit check is bypassed). After the deposit is over, the borrowAToken balances would be:

- borrower = 500ke6 (actually, it would be ~500ke6 due to calculations involving AToken and liquidity index, but, I have used 500ke6 for simplicity).
- Size contract = 0
- Total supply of borrowAToken = 1.49Me6

After the repayment is over, the borrowAToken balances are:

- borrower = 400ke6
- Size contract = 100ke6
- Total supply of borrowAToken = 1.49Me6



Also, after repayment:

```
debtToken.balanceOf[borrower] = 0
```

Now, the above situation breaks the invariant:

```
increase in borrowAToken supply (= 500ke6) is not <= decrease
in debtToken supply (= 100ke6)
```

The multicall flow should revert. However, if we look at the `multicall(State storage state, bytes[] calldata data)` function in `Multicall.sol`:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/Multicall.sol#L26>

```
function multicall(State storage state, bytes[] calldata data)
internal returns (bytes[] memory results) {
    state.data.isMulticall = true;

    uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.balanceOf(address(this));
    uint256 debtTokenSupplyBefore =
state.data.debtToken.totalSupply();

    results = new bytes[](data.length);
    for (uint256 i = 0; i < data.length; i++) {
        results[i] =
Address.functionDelegateCall(address(this), data[i]);
    }

    uint256 borrowATokenSupplyAfter =
state.data.borrowAToken.balanceOf(address(this));
    uint256 debtTokenSupplyAfter =
state.data.debtToken.totalSupply();

    state.validateBorrowATokenIncreaseLteDebtTokenDecrease(
        borrowATokenSupplyBefore, debtTokenSupplyBefore,
        borrowATokenSupplyAfter, debtTokenSupplyAfter
    );

    state.data.isMulticall = false;
}
}
```

LOC-29 sets the `borrowATokenSupplyBefore` variable to

```
borrowAToken.balanceOf(Size_contract) :
```

```
uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.balanceOf(address(this));
```

LOC-37 sets the `borrowATokenSupplyAfter` variable to

```
borrowAToken.balanceOf(Size_contract) :
```

```
uint256 borrowATokenSupplyAfter =
state.data.borrowAToken.balanceOf(address(this));
```

As per our example:

```
uint256 borrowATokenSupplyBefore = 0
uint256 borrowATokenSupplyAfter = 100ke6
```

Now, `validateBorrowATokenIncreaseLteDebtTokenDecrease()` is called at LOC-40:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L19>

```
function validateBorrowATokenIncreaseLteDebtTokenDecrease(
    State storage state,
    uint256 borrowATokenSupplyBefore,
    uint256 debtTokenSupplyBefore,
    uint256 borrowATokenSupplyAfter,
    uint256 debtTokenSupplyAfter
) external view {
    // If the supply is above the cap
    if (borrowATokenSupplyAfter >
state.riskConfig.borrowATokenCap) {
        uint256 borrowATokenSupplyIncrease =
borrowATokenSupplyAfter > borrowATokenSupplyBefore
        ? borrowATokenSupplyAfter -
borrowATokenSupplyBefore
        : 0;
        uint256 debtATokenSupplyDecrease =
debtTokenSupplyBefore > debtTokenSupplyAfter ?
debtTokenSupplyBefore - debtTokenSupplyAfter : 0;

        // and the supply increase is greater than the debt
reduction
        if (borrowATokenSupplyIncrease >
debtATokenSupplyDecrease) {
            // revert
            revert
Errors.BORROW_ATOKEN_INCREASE_EXCEEDS_DEBT_TOKEN_DECREASE(
                borrowATokenSupplyIncrease,
debtATokenSupplyDecrease
            );
        }
    }
}
```

```

    }
    // otherwise, it means the debt reduction was greater
    than the inflow of cash: do not revert
    }
    // otherwise, the supply is below the cap: do not revert
}

```

In this function, the first `if()` statement at LOC-27 would be executed, as per our example:

```
if(100ke6 > 1Me6)
```

... which would be false.

Therefore, the control flow does not move in the `if()` block, and the protocol assumes “the supply is below the cap: do not revert”. As a result, the multicall function would not revert.

The root cause of this issue is the use of `state.data.borrowAToken.balanceOf(address(this))` to set variables `borrowATokenSupplyBefore` and `borrowATokenSupplyAfter`. Instead of `state.data.borrowAToken.balanceOf(address(this))`, `state.data.borrowAToken.totalSupply()` should be used. This would work as then:

```

borrowATokenSupplyBefore would be 990ke6
borrowATokenSupplyAfter would be 1.49Me6

```

And, in the `if()` block of `validateBorrowATokenIncreaseLteDebtTokenDecrease()` function, we would have:

```
if(1.49Me6 > 1Me6)
```

... which would be true. Thus, the control flow would enter into the `if()` block. At LOC-35, we would have:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L35>

```

    // and the supply increase is greater than the debt
    reduction
    if (borrowATokenSupplyIncrease >
    debtATokenSupplyDecrease) {
        // revert
        revert
    }
    Errors.BORROW_ATOKEN_INCREASE_EXCEEDS_DEBT_TOKEN_DECREASE(
        borrowATokenSupplyIncrease,
        debtATokenSupplyDecrease
    )
}

```

```
    );
}
```

That is:

```
if(500ke6 > 100ke6)
```

... which would be true again and finally, the transaction would revert.

This root cause gives rise to one more issue:

- A user would use the multicall for a deposit action only but would be able to bypass the deposit limit.

To comprehend this, think of a similar situation as above where:

- The cap of `borrowAToken supply` = 1Me6
- Current total supply of `borrowAToken` = 990ke6

Moreover, to keep things simple, let's assume that:

```
borrowToken.balanceOf[Size_contract] = 0
```

Now, a user deposits 200ke6 USDC using the multicall function. The normal (that is, without multicall) USDC deposit flow contains a logic to check for the deposit limit, which is bypassed when a multicall is used:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Deposit.sol#L80>

```
if (!state.data.isMulticall) {
    state.validateBorrowATokenCap();
}
```

After completing the deposit, the multicall function would call the

`validateBorrowATokenIncreaseLteDebtTokenDecrease()` function for a check (similar to above).

The following arguments would have the specified value:

```

        uint256 borrowATokenSupplyBefore =
        borrowToken.balanceOf[Size_contract] = 0
        uint256 borrowATokenSupplyAfter =
        borrowToken.balanceOf[Size_contract] = 0

```

Now, again in the `if()` block at LOC-27:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L27>

```

        if (borrowATokenSupplyAfter >
        state.riskConfig.borrowATokenCap)

```

We would have the following:

```

        if(0 > 1Me6)

```

... which would be false, and, so, the control flow would not enter the `if()` block. Thus, the multicall flow would pass with the following result:

- Current total supply of `borrowAToken` = 1.19Me6
- The cap of total supply of `borrowAToken` = 1Me6

The user was successful in depositing USDC in spite of the fact that his deposit crossed the deposit limit.

## Severity

- **Impact:** An invariant, which should not break, is broken. This point sets the impact of this issue to be Medium.
- **Likelihood:** Any user would call the Multicall function (since it is not access-controlled) and can bypass the deposit limit as well as the restriction: increase in `borrowAToken` supply  $\leq$  decrease in `debtToken` supply. This makes the likelihood high.

The final severity comes to be Medium. Moreover, the multicall functionality does not work as intended, affecting the availability of the correct intended version of multicall.

## Recommended Mitigation Steps

Apply the following in `multicall(State storage state, bytes[] calldata data)` function:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/Multicall.sol#L26>

```
function multicall(State storage state, bytes[] calldata data)
internal returns (bytes[] memory results) {
    state.data.isMulticall = true;

    -    uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.balanceOf(address(this));
+    uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.totalSupply();
        uint256 debtTokenSupplyBefore =
state.data.debtToken.totalSupply();

        results = new bytes[](data.length);
        for (uint256 i = 0; i < data.length; i++) {
            results[i] =
Address.functionDelegateCall(address(this), data[i]);
        }

    -    uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.balanceOf(address(this));
+    uint256 borrowATokenSupplyBefore =
state.data.borrowAToken.totalSupply();
        uint256 debtTokenSupplyAfter =
state.data.debtToken.totalSupply();

        state.validateBorrowATokenIncreaseLteDebtTokenDecrease(
            borrowATokenSupplyBefore, debtTokenSupplyBefore,
            borrowATokenSupplyAfter, debtTokenSupplyAfter
        );

        state.data.isMulticall = false;
    }
}
```

## Assessed type

Invalid Validation

[aviggiano \(Size\) confirmed and commented via duplicate Issue #144:](#)

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/126>.*

*Note: For full discussion, see [here](#).*

## 🔗 [M-02] Users can not to buy/sell minimum credit allowed due to exactAmountIn condition

---

Submitted by [iam\\_emptyset](#), also found by [ether\\_sky](#), [Ox04bytes](#), [samuraii77](#), [Jorgetc](#), [radin100](#) (1, 2), [DanielArmstrong](#), [OxStalin](#), [Oxanmol](#), [carlos\\_alegre](#), [inzinko](#), [dthank](#), [Infect3d](#) (1, 2), [hyh](#), [Oxarno](#), [elhaj](#), [zarkk01](#), [trachev](#), [KupiaSec](#), [OxRobocop](#), [ilchovski](#), [OxJoyBoy03](#), [BoltzmannBrain](#), [pkqs90](#), and [said](#)

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/BuyCreditMarket.sol#L91>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/SellCreditMarket.sol#L93>

### 🔗 Impact

The minimum credit user can buy or sell can be gotten from `state.riskConfig.minimumCreditBorrowAToken` which was set as `5e6`. However, in `buyCreditMarket` and `sellCreditMarket` user can not buy or sell minimum credit allowed or small amount above it due to the `exactAmountIn` condition he chose.

In `buyCreditMarket`: when user set `params.exactAmountIn = true`, The `params.amount` value will be cash he want to use to buy credit. Since the `params.amount` value is cash it can be set less than `state.riskConfig.minimumCreditBorrowAToken` (`5e6`) as long as when the value get converted to credit it will reach the `state.riskConfig.minimumCreditBorrowAToken` value. But, unfortunately, in `validateBuyCreditMarket()` whenever `params.amount` is less than `state.riskConfig.minimumCreditBorrowAToken` the transaction will revert due to below code present in the function.

```
if (params.amount < state.riskConfig.minimumCreditBorrowAToken) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}
```

For example, the cash user can use to buy minimum credit allowed (`5e6`) can be calculated as in the code below, And the value should be less than the minimum credit allowed (`5e6`).

```
uint256 minimumCash = Math.mulDivUp(5e6, PERCENT, PERCENT +
ratePerTenor);
```

In `sellCreditMarket` : when user set `params.exactAmountIn = false` , The `params.amount` will be the exact cash he want to receive. But he will also not be able to set the `params.amount` value to be less than `state.riskConfig.minimumCreditBorrowAToken` value due to the below code present in `validateSellCreditMarket()` .

```
if (params.amount < state.riskConfig.minimumCreditBorrowAToken) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}
```

## Proof of Concept

**For** `BuyCreditMarket` : Copy and paste below code in

`/test/local/actions/BuyCreditMarket.t.sol` :

```
function testFail_UserCanNotBuyMinimumCredit() public {
    // alice deposit weth
    _deposit(alice, weth, 100e18);

    // bob deposit usdc
    _deposit(bob, usdc, 200e6);

    // alice create a borrow offer
    _sellCreditLimit(alice, 0.03e18, 365 days);

    // calculate cash from minimum credit borrowAToken allowed =
5e6
    uint256 minimumCash = Math.mulDivUp(5e6, PERCENT, PERCENT +
0.03e18);

    // Expect revert with an error
    CREDIT_LOWER_THAN_MINIMUM_CREDIT
    uint256 debtPositionId = _buyCreditMarket(bob, alice,
minimumCash, 365 days, true);
}
```

Then run the test with below command, the test should pass because it will revert with an error `Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT` .

```
forge test --mt testFail_UserCanNotBuyMinimumCredit
```

**For** `SellCreditMarket` : Copy and paste below code in

`/test/local/actions/SellCreditMarket.t.sol` :



```

function testFail_UserCanNotSellMinimumCredit() public {
    // alice deposit usdc
    _deposit(alice, usdc, 200e6);

    // bob deposit weth
    _deposit(bob, weth, 100e18);

    // alice create a loan offer
    _buyCreditLimit(alice, block.timestamp + 365 days,
YieldCurveHelper.pointCurve(365 days, 0.03e18));

    // calculate cash from minimum credit borrowAToken allowed =
5e6
    uint256 minimumCreditInCash = Math.mulDivUp(5e6, PERCENT,
PERCENT + 0.03e18);

    // Expect revert with an error
CREDIT_LOWER_THAN_MINIMUM_CREDIT
    uint256 debtPositionId = _sellCreditMarket(bob, alice,
RESERVED_ID, minimumCreditInCash, 365 days, false);
}

```

Then run the test with below command, the test should pass because it will revert with an error `Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT`.

```
forge test --mt testFail_UserCanNotSellMinimumCredit
```

## Recommended Mitigation Steps

In `validateBuyCreditMarket()` and `validateSellCreditMarket()` for `BuyCreditMarket` and `SellCreditMarket` libraries respectively, there is a need of considering the condition of `params.exactAmountIn` value. The check should be implemented as shown below.

**For `BuyCreditMarket` library:** inside `validateBuyCreditMarket()` replace below code:

```

if (params.amount < state.riskConfig.minimumCreditBorrowAToken) {
    // @audit 5e6 USDC
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}

```

With the below code:

```

uint256 ratePerTenor =
borrowOffer.getRatePerTenor(VariablePoolBorrowRateParams({variable
PoolBorrowRate: state.oracle.variablePoolBorrowRate,
variablePoolBorrowRateUpdatedAt:
state.oracle.variablePoolBorrowRateUpdatedAt,
variablePoolBorrowRateStaleRateInterval:
state.oracle.variablePoolBorrowRateStaleRateInterval}), tenor);

if (params.exactAmountIn && params.amount <
Math.mulDivUp(state.riskConfig.minimumCreditBorrowAToken, PERCENT,
PERCENT + ratePerTenor)) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
} else if (!params.exactAmountIn && params.amount <
state.riskConfig.minimumCreditBorrowAToken) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}

```

**For SellCreditMarket library:** inside `validateSellCreditMarket()` replace below code:

```

if (params.amount < state.riskConfig.minimumCreditBorrowAToken) {
// @audit 5e6 USDC
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}

```

With the below code:

```

uint256 ratePerTenor =
loanOffer.getRatePerTenor(VariablePoolBorrowRateParams({variablePo
olBorrowRate: state.oracle.variablePoolBorrowRate,
variablePoolBorrowRateUpdatedAt:
state.oracle.variablePoolBorrowRateUpdatedAt,
variablePoolBorrowRateStaleRateInterval:
state.oracle.variablePoolBorrowRateStaleRateInterval}), tenor);

if (params.exactAmountIn && params.amount <
state.riskConfig.minimumCreditBorrowAToken) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
} else if (!params.exactAmountIn && params.amount <
Math.mulDivUp(state.riskConfig.minimumCreditBorrowAToken, PERCENT,
PERCENT + ratePerTenor)) {
    revert Errors.CREDIT_LOWER_THAN_MINIMUM_CREDIT(params.amount,
state.riskConfig.minimumCreditBorrowAToken);
}

```

## Assessed type

Invalid Validation

aviggiano (Size) confirmed and commented:

Fixed in <https://github.com/SizeCredit/size-solidity/pull/125>

## [M-03] Size uses wrong source to query available liquidity on Aave, resulting in borrow and lend operations being bricked upon mainnet deployment

Submitted by [JCN](#), also found by [mt030d](#), [Oxpiken](#), [silver\\_eth](#), [grearlake](#), [elhaj](#), [zarkk01](#), [trachev](#), [ilchovski](#), [bin2chen](#), [VAD37](#), and [SpicyMeatball](#)

Size queries the `underlyingBorrowToken` balance of the `variablePool` during borrowing and lending actions in order to check available liquidity. If there is not enough available liquidity for the new borrower to withdraw the borrowed assets, the function call will revert:

### Size::buyCreditMarket

```
178:     function buyCreditMarket(BuyCreditMarketParams calldata
    params) external payable override(ISize) whenNotPaused {
    ...
184:         state.validateVariablePoolHasEnoughLiquidity(amount);
    // @audit: liquidity check
```

### Size::sellCreditMarket

```
188:     function sellCreditMarket(SellCreditMarketParams memory
    params) external payable override(ISize) whenNotPaused {
    ...
194:         state.validateVariablePoolHasEnoughLiquidity(amount);
    // @audit: liquidity check
```

### CapsLibrary::validateVariablePoolHasEnoughLiquidity

```

67:     function validateVariablePoolHasEnoughLiquidity(State
storage state, uint256 amount) public view {
68:         uint256 liquidity =
state.data.underlyingBorrowToken.balanceOf(address(state.data.vari
ablePool)); // @audit: ATokens hold liquidity, not variablePool
69:         if (liquidity < amount) {
70:             revert
Errors.NOT_ENOUGH_BORROW_ATHEN_LIQUIDITY(liquidity, amount);
71:         }

```

However, Aave actual holds all liquidity in the AToken contracts, not the Pool contracts. Therefore, the liquidity check above will always fail when the `variablePool` references a live instance of an actual Aave Pool. This is due to the fact that the `variablePool` balance will be 0 (does not hold any liquidity), causing borrowing and lending actions to revert on line 70 in `CapsLibrary.sol`.

The below code snippets are taken from out of scope contracts, but are shown to further explain why the test suite did not detect this vulnerability:

### SupplyLogic::executeSupply

```

52:     function executeSupply(
...
67:         IERC20(params.asset).safeTransferFrom(msg.sender,
reserveCache.aTokenAddress, params.amount);

```

As shown above, the actual implementation of the Aave pool will transfer supplied assets to the `aTokenAddress` on supplies. The Aave pool implementation will then transfer the assets from the AToken to the recipient during withdrawals, since the liquidity is stored in the AToken contract:

### SupplyLogic::executeWithdraw

```

106:     function executeWithdraw(
...
139:         IAToken(reserveCache.aTokenAddress).burn(
140:             msg.sender,
141:             params.to,
142:             amountToWithdraw,
143:             reserveCache.nextLiquidityIndex
144:         );

```

### AToken::burn

```

96:     function burn(
97:         address from,
98:         address receiverOfUnderlying,
99:         uint256 amount,
100:        uint256 index
101:    ) external virtual override onlyPool {
102:        _burnScaled(from, receiverOfUnderlying, amount, index);
103:        if (receiverOfUnderlying != address(this)) {
104:
IERC20(_underlyingAsset).safeTransfer(receiverOfUnderlying,
amount);

```

However, Size uses a `PoolMock` contract for their test suite, which implements logic that differs from actual Aave Pool contracts:

### **PoolMock.sol#L67-L78**

```

67:     function supply(address asset, uint256 amount, address
onBehalfOf, uint16) external {
68:         Data memory data = datas[asset];
69:         IERC20Metadata(asset).transferFrom(msg.sender,
address(this), amount); // @audit: underlying transferred from
supplier to PoolMock contract
70:         data.aToken.mint(address(this), onBehalfOf, amount,
data.reserveIndex);
71:     }
72:
73:     function withdraw(address asset, uint256 amount, address
to) external returns (uint256) {
74:         Data memory data = datas[asset];
75:         data.aToken.burn(msg.sender, address(data.aToken),
amount, data.reserveIndex); // @audit: no assets transferred from
AToken
76:         IERC20Metadata(asset).safeTransfer(to, amount); //
@audit: underlying transferred from PoolMock contract to supplier
77:         return amount;
78:     }

```

As shown above, the `PoolMock` contract transfers all supplied assets to the `PoolMock` contract itself, not the `AToken` contract. This differs from how actual Aave Pools handle liquidity on live networks.

## Impact

Borrow ( `sellCreditMarket` ) and lend ( `buyCreditMarket` ) operations will be bricked when Size is deployed and integrates with Aave on a live network. This report is labeled as High due to

the fact that the main functionalities of this lending protocol will be unusable upon deployment.

## Proof of Concept

The following PoC simply showcases the fact that the `AToken` does not hold the liquidity in the test suite. See [here](#) to observe that `ATokens` are the contracts that hold liquidity for Aave in live networks.

Modify the file below and run with `forge test --mc BuyCreditLimitTest --mt test_BuyCreditLimit_buyCreditLimit_adds_loanOffer_to_orderbook :`

```
diff --git a./test/local/actions/BuyCreditLimit.t.sol
b./test/local/actions/BuyCreditLimit.t.sol
index a8c6317..66dd63e 100644
--- a./test/local/actions/BuyCreditLimit.t.sol
+++ b./test/local/actions/BuyCreditLimit.t.sol
@@ -23,6 +23,9 @@ contract BuyCreditLimitTest is BaseTest {
    assertTrue(_state().alice.user.loanOffer.isNull());
    _buyCreditLimit(alice, block.timestamp + 12 days,
YieldCurveHelper.pointCurve(12 days, 1.01e18));
    assertTrue(!_state().alice.user.loanOffer.isNull());
+    address aToken =
variablePool.getReserveData(address(usdc)).aTokenAddress;
+    assertEquals(usdc.balanceOf(address(variablePool)), 100e6);
+    assertEquals(usdc.balanceOf(aToken), 0);
    }
```

## Recommended Mitigation

When checking available liquidity on Aave, Size should query the `underlyingBorrowToken` balance of the `AToken` instead of the `variablePool` :

```
diff --git a./src/libraries/CapsLibrary.sol
b./src/libraries/CapsLibrary.sol
index 7e35e90..b7ce7a1 100644
--- a./src/libraries/CapsLibrary.sol
+++ b./src/libraries/CapsLibrary.sol
@@ -65,9 +65,11 @@ library CapsLibrary {
    /// @param state The state struct
    /// @param amount The amount of cash to withdraw
    function validateVariablePoolHasEnoughLiquidity(State storage
state, uint256 amount) public view {
-    uint256 liquidity =
state.data.underlyingBorrowToken.balanceOf(address(state.data.vari
ablePool));
```

```
+         address aToken =  
state.data.variablePool.getReserveData(address(state.data.underlyi  
ngBorrowToken)).aTokenAddress;  
+         uint256 liquidity =  
state.data.underlyingBorrowToken.balanceOf(aToken);  
         if (liquidity < amount) {  
             revert  
Errors.NOT_ENOUGH_BORROW_ATOKEN_LIQUIDITY(liquidity, amount);  
         }  
    }  
}
```

**aviggiano (Size) confirmed and commented:**

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/127>*

**samurai77 (warden) commented:**

*Hi, the severity of this issue should be a medium. Please take a look at what a high severity issue should be:*

*Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).*

*This issue does not cause such an issue. In fact, it can be completely mitigated by redeploying the contract or upgrading the contract (if it is upgradeable, I am not sure if it was) whenever the issue is noticed which would be extremely quickly as it would occur on the first deposit of USDC essentially causing nothing else than some wasted time and headache.*

**hansfrieze (judge) decreased severity to Medium and commented:**

*I agree. Medium is more appropriate as there is no direct fund loss.*

## [M-04] Inadequate checks to confirm the correct status of the sequence/ `sequencerUptimeFeed` in `PriceFeed.getPrice()` contract

Submitted by [adeolu](#), also found by [serial-coder](#)

The `PriceFeed` contract has `sequencerUptimeFeed` checks in place to assert if the sequencer on an L2 is running but these checks are not implemented correctly. The [chainlink docs](#) say that `sequencerUptimeFeed` can return a 0 value for `startedAt` if it is called during an “invalid round”.

**startedAt:** *This timestamp indicates when the sequencer changed status. This timestamp returns 0 if a round is invalid. When the sequencer comes back up after an outage, wait for the `GRACE_PERIOD_TIME` to pass before accepting answers from the data feed. Subtract `startedAt` from `block.timestamp` and revert the request if the result is less than the `GRACE_PERIOD_TIME`.*

*If the sequencer is up and the `GRACE_PERIOD_TIME` has passed, the function retrieves the latest answer from the data feed using the `dataFeed` object.*

Please note that an “invalid round” is described to mean there was a problem updating the sequencer’s status, possibly due to network issues or problems with data from oracles, and is shown by a `startedAt` time of 0 and `answer` is 0. Further explanation can be seen as given by an official chainlink engineer as seen here in the chainlink public discord

*Note: to view the provided image, please see the original submission [here](#).*

This makes the implemented check below in the `PriceFeed.getPrice()` to be useless if its called in an invalid round.

```

        if (block.timestamp - startedAt <= GRACE_PERIOD_TIME)
        {
            revert Errors.GRACE_PERIOD_NOT_OVER();
        }

```

As `startedAt` will be 0, the arithmetic operation `block.timestamp - startedAt` will result in a value greater than `GRACE_PERIOD_TIME` (which is hardcoded to be 3600). I.e., `block.timestamp` = 1719739032, so 1719739032 - 0 = 1719739032 which is bigger than 3600. The code won’t revert.



Imagine a case where a round starts, at the beginning `startedAt` is recorded to be 0, and `answer`, the initial status is set to be 0. Note that docs say that if `answer = 0`, sequencer is up, if equals to 1, sequencer is down. But in this case here, `answer` and `startedAt` can be 0 initially, until after all data is gotten from oracles and update is confirmed then the values are reset to the correct values that show the correct status of the sequencer.

From these explanations and information, it can be seen that `startedAt` value is a second value that should be used in the check for if a sequencer is down/up or correctly updated. The checks in `PriceFeed.getPrice()` will allow for successful calls in an invalid round because reverts don't happen if `answer == 0` and `startedAt == 0` thus defeating the purpose of having a `sequencerFeed` check to ascertain the status of the `sequencerFeed` on L2 (i.e., if it is up/down/active or if its status is actually confirmed to be either).

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/oracle/PriceFeed.sol#L68C1-L76C14>

```

        if (answer == 1) {
            // sequencer is down
            revert Errors.SEQUENCER_DOWN();
        }

        if (block.timestamp - startedAt <= GRACE_PERIOD_TIME)
        {
            // time since up
            revert Errors.GRACE_PERIOD_NOT_OVER();
        }

```

## Impact

Inadequate checks to confirm the correct status of the sequencer/ `sequencerUptimeFeed` in `PriceFeed.getPrice()` contract will cause `getPrice()` to not revert even when the sequencer uptime feed is not updated or is called in an invalid round.

## Recommended Mitigation Steps

Add a check that reverts if `startedAt` is returned as 0.

## Assessed type

Oracle

[aviggiano \(Size\) confirmed and commented:](#)

*This looks valid. It's weird that the Chainlink sample code does not implement the check suggested by the documentation, as the warden points out.*

Fixed in <https://github.com/SizeCredit/size-solidity/pull/140>.

CC: <https://github.com/smartcontractkit/documentation/pull/1995>.

## 🔗 [M-05] Users may incur an unexpected fragmentation fee in the `compensate()` call

Submitted by [mt030d](#), also found by [pkqs90](#)

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/Compensate.sol#L116>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/Compensate.sol#L136>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/Compensate.sol#L146-L155>

## 🔗 Impact

The `CompensateParams` struct lacks a field indicating the minimum amount the user is willing to compensate in the `compensate()` call. Consequently, users might pay an unexpected fragmentation fee during the `compensate()` call.

In scenarios where a borrower has a credit position that can be used to compensate the debt (`creditPositionToCompensateId`), they could search the market for a credit position (`creditPositionWithDebtToRepayId`) that relates to the debt and has the same credit amount as `creditPositionToCompensateId`. This way, in the `compensate()` call, the borrower would not fragment the `creditPositionToCompensateId` and avoid paying the fragmentation fee.

However, unexpectedly to the borrower, they might still pay the fragmentation fee if the `compensate()` transaction is executed after a `buyCreditMarket()` or `sellCreditMarket()` transaction that decreases the credit in `creditPositionWithDebtToRepayId`.

```
uint256 amountToCompensate = Math.min(params.amount,
    creditPositionWithDebtToRepay.credit);
```

```

        uint256 exiterCreditRemaining =
creditPositionToCompensate.credit - amountToCompensate;

        if (exiterCreditRemaining > 0) {
            // charge the fragmentation fee in collateral
tokens, capped by the user balance
            uint256 fragmentationFeeInCollateral = Math.min(

state.debtTokenAmountToCollateralTokenAmount(state.feeConfig.fragm
entationFee),

state.data.collateralToken.balanceOf(msg.sender)
        );
        state.data.collateralToken.transferFrom(
            msg.sender, state.feeConfig.feeRecipient,
fragmentationFeeInCollateral
        );
    }

```

In this case, the `amountToCompensate` will be less than the credit in `creditPositionToCompensateId`, and the user will have to pay an unforeseen fragmentation fee.

This situation can occur in normal user flows. Moreover, a malicious user could exploit this vulnerability and front-running a borrower's `compensate()` transaction to cause the borrower to pay an unforeseen fragmentation fee.

## Proof of Concept

```

// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import {NonTransferrableToken} from
"@src/token/NonTransferrableToken.sol";
import {CREDIT_POSITION_ID_START} from
"@src/libraries/LoanLibrary.sol";
import {DataView} from "@src/SizeView.sol";
import {BaseTest} from "@test/BaseTest.sol";
import {console} from "forge-std/console.sol";

contract PoC is BaseTest {
    NonTransferrableToken collateralToken;

    function setUp() public override {
        super.setUp();
        _labels();
    }

```

```

DataView memory dataView = size.data();
collateralToken = dataView.collateralToken;
vm.label(address(collateralToken), "szWETH");

_deposit(alice, weth, 100e18);
_deposit(alice, usdc, 100e6);
_deposit(bob, weth, 100e18);
_deposit(bob, usdc, 100e6);
_deposit(candy, weth, 100e18);
_deposit(candy, usdc, 100e6);
_deposit(james, weth, 100e18);
_deposit(james, usdc, 100e6);
}

function test_it() public {

    _sellCreditLimit(alice, 0.03e18, 365 days);
    _buyCreditMarket(bob, alice, 100e6, 365 days, false);

    (, uint256 creditPositionCount) =
size.getPositionsCount();
    uint256 creditPositionWithDebtToRepayId =
CREDIT_POSITION_ID_START + creditPositionCount - 1;

    _sellCreditLimit(candy, 0.04e18, 365 days);
    _buyCreditMarket(alice, candy, 100e6, 365 days, false);
    (, creditPositionCount) = size.getPositionsCount();
    uint256 creditPositionToCompensateId =
CREDIT_POSITION_ID_START + creditPositionCount - 1;

    uint256 snapshot = vm.snapshot();
    {
        uint256 beforeBalance =
collateralToken.balanceOf(alice);
        _compensate(alice, creditPositionWithDebtToRepayId,
creditPositionToCompensateId);
        uint256 afterBalance =
collateralToken.balanceOf(alice);
        console.log("Scenario 1: borrower's expected
compensate call");
        console.log("  fragmentation fee: %s", beforeBalance -
afterBalance);
    }

    vm.revertTo(snapshot);
    _sellCreditLimit(bob, 0.04e18, 365 days);
    _buyCreditMarket(james, creditPositionWithDebtToRepayId,
50e6, false);
    {
        uint256 beforeBalance =
collateralToken.balanceOf(alice);
        _compensate(alice, creditPositionWithDebtToRepayId,
creditPositionToCompensateId);

```

```
        uint256 afterBalance =
        collateralToken.balanceOf(alice);
        console.log("Scenario 2: unexpected - a
        buyCreditMarket call executed before the compensate call");
        console.log("  fragmentation fee: %s", beforeBalance -
        afterBalance);
    }
}
}
```

Run the above code with command `forge test --mc PoC -vv` , the result is as follows:

Logs:

```
Scenario 1: borrower's expected compensate call
  fragmentation fee: 0
Scenario 2: unexpected - a buyCreditMarket call executed
before the compensate call
  fragmentation fee: 3739715781600599
```

As show by the PoC, by the time the `compensate` transaction is executed in scenario 2, the credit in `creditPositionWithDebtToRepayId` has decreased due to an earlier `buyCreditMaket` transaction. As a result, the borrower pays an unforeseen fragmentation fee.

## Tools Used

Foundry

## Recommended Mitigation Steps

Allow the user to input a `minAmount` parameter in the `compensate()` function to specify the minimum amount they are willing to use. Then, handle this parameter within the `compensate()` function.

**aviggiano (Size) acknowledged and commented:**

*Additional feedback from the team:*

*I think this is technically valid since, although frontrunning can happen, we can implement measures in the protocol to prevent users from being unexpectedly charged fees. For example, setting slippage protection. compensate could have a parameter allowing the user to specify “revert if a fee is charged”.*

*Yes, it is another instance of having a proper Anti-MEV system because the same “issue” is true on the other side, meaning somebody trying to buy a credit*

*that gets compensated entirely before his `buyCreditMarket()` can be executed but on buy and sell we already Anti-MEV params. That's kind of minor since no attack that benefits anybody, but the protocol, can be run here so up to you.*

*I think the potential issue of users paying for fees unexpectedly is not as important as the UX problem of operations failing because of the order they are applied to the orderbook, as we are aware. So adding a new parameter to the function doesn't hurt but doesn't solve the problem either.*

hansfrieze (judge) commented:

*Will keep as Medium since it might occur during normal interactions.*

🔗 **[M-06] Neither `sellCreditMarket()`, nor `compensate()`, checks whether the credit position to be sold is allowed for sale**

---

Submitted by Oxpiken, also found by adeolu, hyh, and Shield

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/SellCreditMarket.sol#L51-L122>  
<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Compensate.sol#L42-L101>

## 🔗 Impact

A credit buyer could be forced to buy a not for sale credit position with no way to sell it passively, potentially suffering a loss due to missing an arbitrage opportunity.

## 🔗 Proof of Concept

Any user is allowed to lend or borrow `szaUSDC` through `Size` protocol.

- A borrower can create a limit borrowing order by calling `sellCreditLimit()`, anyone can lend their `szaUSDC` to this borrower by calling `buyCreditMarket()`.

- A lender can create a limit lending order by calling `buyCreditLimit()` , anyone can borrow `szaUSDC` from this lender by calling `sellCreditMarket()` .
- A existing credit position can be sold by its owner by `sellCreditMarket()` by default.
- A existing credit position by default could be bought by another borrower as long as the owner of credit position has created a limit borrowing order.

A lender can set their `allCreditPositionsForSaleDisabled` to `true` , which prevents any of their credit positions from being sold. Alternatively, a specific credit position can be marked as `not for sale` , making it impossible to sell and for anyone to buy it.

While no one can buy a `not for sale` credit position, their owner can force to sell it or use it to compensate their debt. Once sold, no one can buy it again due to its `not for sale` status, resulting the new owner potentially suffering a loss.

Copy below codes to **SellCreditMarketTest.t.sol** and run `forge test --match-test test_SellCreditMarket_sellNotForSaleCreditPosition` :

## ► Details

As we can see, as a speculator, Candy might suffer a loss because the credit position she is forced to buy is not able to be bought from her by any other buyer.

## 🔗 Recommended Mitigation Steps

No one should be allowed to sell their `not for sale` credit positions or use them to compensate their debts:

## ► Details

## 🔗 Assessed type

Context

**aviggiano (Size) confirmed and commented:**

*This was not very well specified in our documentation, so it might be an issue.*

*On the one hand, not enforcing `forSale` for credit owners is good because we do not want to force users to do 2 actions if they are the ones selling them. On the other hand, passive buyers can end up with credits that can't be resold.*

*Additional feedback from the team:*

*I think we should reset the variable upon sale.*

*So I think this issue is technically valid.*

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/130>.*

**hansfrieze (judge) commented:**

*The root cause is that `sellCreditMarket()` and `compensate()` don't check `forSale` flag of a credit position. (`creditPositionToCompensateId` in `compensate()`).*

*There are two impacts:*

- The new lender would have a credit position of `forSale = false` In case of a full sale. ([#184](#), [#335](#))
- The credit position's `forSale` would be changed from false to true in case of a partial sale. ([#228](#))

*Will consider them as duplicates because they have the same root cause and can be mitigated by resetting the flag consistently upon sale.*

*Note: For full discussion, see [here](#).*

## [🔗](#) **[M-07] Credit can be sold forcibly as `forSale` setting can be ignored via Compensate**

Submitted by [hyh](#), also found by [OxAlix2](#), [almurhasan](#), [BenRai](#), [OxStalin](#), and [bin2chen](#)

Any credit position can be forcibly sold with the help of its borrower. This will be executed only when have expected profit; for example, when lender's curve is not null and is above the market it is profitable to buy credit from them (lend to them at above market rates), but they might block it with the lack of free collateral and the `forSale = false` flag, which can be overridden by the corresponding borrower of this credit position. The impact of this forced



sale is proportional to interest rate volatility and can be substantial. There are no additional prerequisites for the setup.

## Proof of Concept

`createDebtAndCreditPositions` creates credit positions with `forSale == true`:

### AccountingLibrary.sol#L62-L82

```
function createDebtAndCreditPositions(
    ...
) external returns (CreditPosition memory creditPosition) {
    DebtPosition memory debtPosition =
        DebtPosition({borrower: borrower, futureValue:
futureValue, dueDate: dueDate, liquidityIndexAtRepayment: 0});

    uint256 debtPositionId = state.data.nextDebtPositionId++;
    state.data.debtPositions[debtPositionId] = debtPosition;

    emit Events.CreateDebtPosition(debtPositionId, borrower,
lender, futureValue, dueDate);

    creditPosition = CreditPosition({
        lender: lender,
        credit: debtPosition.futureValue,
        debtPositionId: debtPositionId,
>>        forSale: true
    });
```

This can be done on demand for any credit position by the borrower of the corresponding debt position, by running `Compensate` with `params.creditPositionToCompensateId == RESERVED_ID`, which will create new position with `forSale == true` and substitute the existing lender position with it:

### Compensate.sol#L118-L145

```
CreditPosition memory creditPositionToCompensate;
if (params.creditPositionToCompensateId == RESERVED_ID) {
>>    creditPositionToCompensate =
state.createDebtAndCreditPositions({ // @audit create new credit
with `forSale == true`
    lender: msg.sender,
    borrower: msg.sender,
    futureValue: amountToCompensate,
    dueDate: debtPositionToRepay.dueDate
});
} else {
    creditPositionToCompensate =
state.getCreditPosition(params.creditPositionToCompensateId);
```

```

        amountToCompensate = Math.min(amountToCompensate,
creditPositionToCompensate.credit);
    }

    // debt and credit reduction
    state.reduceDebtAndCredit(
        creditPositionWithDebtToRepay.debtPositionId,
params.creditPositionWithDebtToRepayId, amountToCompensate
    );

    uint256 exiterCreditRemaining =
creditPositionToCompensate.credit - amountToCompensate;

    // credit emission
>> state.createCreditPosition({
        exitCreditPositionId:
params.creditPositionToCompensateId == RESERVED_ID // @audit give
it to the lender instead of the existing position with `forSale ==
false`

        ? state.data.nextCreditPositionId - 1
        : params.creditPositionToCompensateId,
        lender: creditPositionWithDebtToRepay.lender,
        credit: amountToCompensate
    });

```

Then, it can be then bought with `BuyCreditMarket` :

### BuyCreditMarket.sol#L79-L8

```

        borrower = creditPosition.lender;
        tenor = debtPosition.dueDate - block.timestamp; //
positive since the credit position is transferrable, so the loan
must be ACTIVE
    }

    BorrowOffer memory borrowOffer =
state.data.users[borrower].borrowOffer;

```

## Recommended Mitigation Steps

Consider passing the flag to `createDebtAndCreditPositions()` indicating `forSale` flag to be set, which be passed from the existing credit in [Compensate.sol#L139](#), so it won't be changed.

aviggiano (Size) confirmed and commented:

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/130>.*

**hansfrieze (judge) decreased severity to Medium and commented:**

*Valid finding.*

*Medium is more appropriate due to the below reasons.*

- Buying the lender's credit position doesn't mean any loss to him as it's sold with his borrow offer. (It's just an unintended behavior.)
- The lender can prevent this by setting `allCreditPositionsForSaleDisabled = false`.

## **[M-08] Sandwich attack on loan fulfillment will temporarily prevent users from accessing their borrowed funds**

*Submitted by [gesha17](#), also found by [grearlake](#), [samurair77](#), [mt030d](#), [KupiaSec](#), and [VAD37](#)*

The protocol does not remove borrowed token liquidity from the aave pool once a loan is initiated. It only checks if there is enough available liquidity via the `validateVariablePoolHasEnoughLiquidity()` function. A user is expected to withdraw their funds after the loan is created. This means that a malicious user can sandwich loans that exceed the available liquidity in aave by depositing liquidity in a frontrun transaction and removing it in a backrun transaction. This way a user that takes out a loan will not be able to remove their funds until there is enough available liquidity in the pool.

Also note that liquidity may become unavailable via other factors, like external users removing liquidity from aave.

## **Proof of Concept**

The `sellCreditMarket()` function does not remove liquidity from aave, but only checks if there is enough liquidity available via a call to `validateVariablePoolHasEnoughLiquidity()`.

**<https://github.com/code-423n4/2024-06-size/blob/main/src/Size.sol#L194>**

```
function sellCreditMarket(SellCreditMarketParams memory
params) external payable override(ISize) whenNotPaused {
    state.validateSellCreditMarket(params);
    uint256 amount = state.executeSellCreditMarket(params);
```

```

        if (params.creditPositionId == RESERVED_ID) {

            state.validateUserIsNotBelowOpeningLimitBorrowCR(msg.sender);
        }
        // @audit validate liquidity after execution??
        state.validateVariablePoolHasEnoughLiquidity(amount);
    }

```

Also, the `executeSellCreditMarket()` function transfers the size protocol accounting `borrowAToken` to the borrower, instead of the underlying token itself, leaving the user to withdraw their borrowed funds in a separate transaction.

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/SellCreditMarket.sol#L202>

```

function validateSellCreditMarket(State storage state,
SellCreditMarketParams calldata params) external view {
    ...
    state.data.borrowAToken.transferFrom(params.lender,
msg.sender, cashAmountOut);
    state.data.borrowAToken.transferFrom(params.lender,
state.feeConfig.feeRecipient, fees);
}

```

The `validateVariablePoolHasEnoughLiquidity()` function only checks the balance of the variable pool, and does not remove liquidity:

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L67>

```

function validateVariablePoolHasEnoughLiquidity(State storage
state, uint256 amount) public view {
    uint256 liquidity =
state.data.underlyingBorrowToken.balanceOf(address(state.data.vari
ablePool));
    if (liquidity < amount) {
        revert
Errors.NOT_ENOUGH_BORROW_ATOKEN_LIQUIDITY(liquidity, amount);
    }
}

```

## Recommended Mitigation Steps

Remove the liquidity from the pool when the loan is created and still let the user pull their funds in a separate transaction to remain compliant with pull over push pattern.

[aviggiano \(Size\) acknowledged and commented:](#)

*We acknowledge this issue, but we understand that MEV attacks like frontrunning, sandwich attacks, etc., do not belong to the protocol layer. For example, the on-chain Uniswap logic can't do anything against sandwich attacks.*

*Also, this can be mitigated with a `multicall`.*

*I am not sure how Code4rena judges will weigh in on this issue, but since it was not explicitly stated on the list of Known Limitations, we have decided to make it more explicit in our documentation [here](#).*

### **hansfrieze (judge) commented:**

*Will keep as Medium since it accurately highlights the irrationality of the current validation process.*

## **[M-09] Borrower is not able to compensate his lenders if he is underwater**

Submitted by [ilchovski](#), also found by [OxAlix2](#), [alix40](#), [dhank](#), [zzebra83](#), [ether\\_sky](#), and [pkqs90](#)

A borrower of a loan could have other credit positions with borrowers that have healthy CRs which he could use to compensate his lenders to avoid liquidations and improve his CR. However, he is not able to do this via `compensate()` and he is forced to sell his credit positions via `sellCreditMarket()` or `sellCreditLimit()`. The complications of this are described in the example below.

### **Proof of Concept**

Before we start just to have context, `repay()` repays the whole debt amount of the loan in full and can be called even when the borrower is underwater or past due date.

`compensate()` is used to either split a loan in smaller parts in order to be more easily repaid with `repay()` or to repay certain lenders and reduce the debt of the loan by the borrower giving specific lenders some of his own healthy credit positions that he owns.

Here is an example of the issue: Bob has collateral X in place and has 10 different debt positions (he borrowed). With the newly acquired borrow tokens he buys 10 different credit positions.

Some time passes and his collateral value drops significantly (volatile market), his CR is below the healthy threshold. At the moment Bob does not have any more borrow tokens available but he has 10 healthy credit positions that he can use to improve his CR by compensating some of his lenders.

Here comes the important part: He can successfully compensate 1 of his lenders if at the end of the `compensate(.)` tx his CR becomes healthy. The problem appears when he must compensate for 3 out of all 10 debt positions (more than 1 position) in order to become with healthy CR.

He will call compensate for the first time and the end of this tx he will face a revert due to this check because he needs to compensate 2 more times to become healthy.

### [Link to code](#)

```
function compensate(CompensateParams calldata params) external
payable override(ISize) whenNotPaused {
    state.validateCompensate(params);
    state.executeCompensate(params);
    @> state.validateUserIsNotUnderwater(msg.sender);
}
```

### [Link to code](#)

```
function isUserUnderwater(State storage state, address
account) public view returns (bool) {
    @> return collateralRatio(state, account) <
state.riskConfig.crLiquidation;
}

function validateUserIsNotUnderwater(State storage state,
address account) external view {
    @> if (isUserUnderwater(state, account)) {
        revert Errors.USER_IS_UNDERWATER(account,
collateralRatio(state, account));
    }
}
```

The only option Bob has currently is to sell some of his credit positions via `sellCreditMarket` or `sellCreditLimit`. There might not be any buyers at the moment or he might be forced to take a very bad deal for his credit positions because he will be in a rush in order to repay part of his loans on time to not get liquidated.

## 🔗 Recommended Mitigation Steps

The CR check at the end of compensate is important because fragmentation fees could occur and lower the CR and make it under the healthy threshold in some specific situations.

What I propose as a solution is measuring the CR before and after the `compensate()` logic and it should revert if the CR is becoming worse.

This way Bob will be able to improve his CR by using his credit positions even if he has to compensate multiple times before becoming healthy again.

aviggiano (Size) confirmed and commented:

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/129>*

🔗 **[M-10] withdraw() users may can't withdraw underlyingBorrowToken properly**

*Submitted by [bin2chen](#), also found by [OxAlix2](#), [3n0ch](#), [samuraii77](#), [ayden](#), [inzinko](#), [zarkk01](#), [elhaj](#), [Bob](#), [alix40](#), and [ether\\_sky](#)*

We can withdraw `underlyingCollateralToken` and `underlyingBorrowToken` by `withdraw()`:

```
function withdraw(WithdrawParams calldata params) external
payable override(ISize) whenNotPaused {
    state.validateWithdraw(params);
    state.executeWithdraw(params);
}

@>
state.validateUserIsNotBelowOpeningLimitBorrowCR(msg.sender);
}

function executeWithdraw(State storage state, WithdrawParams
calldata params) public {
    uint256 amount;
    if (params.token ==
address(state.data.underlyingBorrowToken)) {
        amount = Math.min(params.amount,
state.data.borrowAToken.balanceOf(msg.sender));
        if (amount > 0) {
            state.withdrawUnderlyingTokenFromVariablePool(msg.sender,
params.to, amount);
        }
    }
}
```

```

        } else {
            amount = Math.min(params.amount,
state.data.collateralToken.balanceOf(msg.sender));
            if (amount > 0) {

state.withdrawUnderlyingCollateralToken(msg.sender, params.to,
amount);

            }
        }

        emit Events.Withdraw(params.token, params.to, amount);
    }
}

```

From the code above we know that whether we take `underlyingCollateralToken` or `underlyingBorrowToken` will check `validateUserIsNotBelowOpeningLimitBorrowCR()`  $\Rightarrow$  `collateralRatio() > openingLimitBorrowCR`.

This makes sense for taking `underlyingCollateralToken`, but not for taking `underlyingBorrowToken`.

1. Taking the `underlyingBorrowToken` does not affect the `collateralRatio`.
2. The user has already borrowed the funds (with interest accrued and collateralized), it is the user's asset, and should be able to be withdrawn at will, even if it may be liquidated.
3. `openingLimitBorrowCR` is still far from being liquidated, and should not restrict the user from withdrawing the borrowed token.

## 🔗 Impact

If the token is already borrowed, just stored in `Size` and not yet taken, but due to a slight price fluctuation, `validateUserIsNotBelowOpeningLimitBorrowCR()` fails but is still far from being liquidated, the user may not be able to take the borrowed token, resulting in the possibility that the user may not be able to withdraw the borrowed funds.

## 🔗 Recommended Mitigation

```

function withdraw(WithdrawParams calldata params) external
payable override(ISize) whenNotPaused {
    state.validateWithdraw(params);
    state.executeWithdraw(params);
+    if (params.token !=
address(state.data.underlyingBorrowToken) {
+
state.validateUserIsNotBelowOpeningLimitBorrowCR(msg.sender);
}
}

```



```
+      }  
    }
```

## Assessed type

### Context

aviggiano (Size) confirmed and commented:

*This is a valid bug and the report is good.*

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/124>.*

samurai77 (warden) commented:

*Hi, this issue should be of high severity. The likelihood is very high, imagine a user borrows \$1000 and gives \$1500 as collateral; his collateral ratio is now  $1.5e18$ . If the price of ETH drops by just 1 cent (or even less), his borrowed funds will be locked. The impact is also high as well.*

hansfrieze (judge) commented:

*I still believe Medium is appropriate for the following reasons:*

- The user can withdraw the borrowed funds at the time of borrowing using a multicall.
- The user can still use the borrowed funds to repay their loan.
- There is no fund loss; the funds are simply locked for a while.

🔗 [M-11] LiquidateWithReplacement does not charge swap fees on the borrower

---

Submitted by [pkqs90](#), also found by [OxRobocop](#), [BenRai](#), [Brenzee](#), [OxStalin](#), [alix40](#), [3n0ch](#), [Honour](#), [silver\\_eth](#), [trachev](#), [Nyx](#), [KupiaSec](#), [ether\\_sky](#), and [VAD37](#)

When a user places a credit sell limit order, it may be either:

1. Bought by other users.
2. Bought by the `LiquidateWithReplacement` action.

However, for case number 2, it does not charge a swap fee for the borrower, which contradicts to the swap fee definition.

## 🔗 Bug Description

Let's first see how `LiquidateWithReplacement` works. There are two steps:

1. Liquidation: Liquidator sends the amount of debt `debt.futureValue` to the protocol, and receive collateral tokens as liquidation reward.
2. Buy credit: Use the original lender's `debt.futureValue` as credit to buy credit from the borrower. This way the original lender can still get the same amount of credit at the same `dueDate`. The liquidator can then take away the difference between `debt.futureValue` and the amount sent to the borrower.

Note that step 2 is the same as buying credit using the `BuyCreditMarket` function.

However, according to the [docs](#), all cash and credit swaps should charge swap fees on the borrower. The issue here is the swap in `LiquidateWithReplacement` does not charge swap fees.

*Protocol fee; charged to the recipient of USDC on `cash->credit` and `credit->cash` swaps. Prorated based on credit tenor (the fee is annualized).*

```
function executeLiquidateWithReplacement(State storage state,
LiquidateWithReplacementParams calldata params)
    external
    returns (uint256 issuanceValue, uint256
liquidatorProfitCollateralToken, uint256
liquidatorProfitBorrowToken)
{
```

```

emit
Events.LiquidateWithReplacement(params.debtPositionId,
params.borrower, params.minimumCollateralProfit);

    DebtPosition storage debtPosition =
state.getDebtPosition(params.debtPositionId);
    DebtPosition memory debtPositionCopy = debtPosition;
    BorrowOffer storage borrowOffer =
state.data.users[params.borrower].borrowOffer;
    uint256 tenor = debtPositionCopy.dueDate -
block.timestamp;

    liquidatorProfitCollateralToken = state.executeLiquidate(
        LiquidateParams({
            debtPositionId: params.debtPositionId,
            minimumCollateralProfit:
params.minimumCollateralProfit
        })
    );

    uint256 ratePerTenor = borrowOffer.getRatePerTenor(
        VariablePoolBorrowRateParams({
            variablePoolBorrowRate:
state.oracle.variablePoolBorrowRate,
            variablePoolBorrowRateUpdatedAt:
state.oracle.variablePoolBorrowRateUpdatedAt,
            variablePoolBorrowRateStaleRateInterval:
state.oracle.variablePoolBorrowRateStaleRateInterval
        }),
        tenor
    );
    > issuanceValue =
Math.mulDivDown(debtPositionCopy.futureValue, PERCENT, PERCENT +
ratePerTenor);
    liquidatorProfitBorrowToken = debtPositionCopy.futureValue
- issuanceValue;

    debtPosition.borrower = params.borrower;
    debtPosition.futureValue = debtPositionCopy.futureValue;
    debtPosition.liquidityIndexAtRepayment = 0;

    emit Events.UpdateDebtPosition(
        params.debtPositionId,
        debtPosition.borrower,
        debtPosition.futureValue,
        debtPosition.liquidityIndexAtRepayment
    );

    state.data.debtToken.mint(params.borrower,
debtPosition.futureValue);
    > state.data.borrowAToken.transferFrom(address(this),
params.borrower, issuanceValue);

```

```

state.data.borrowAToken.transferFrom(address(this),
state.feeConfig.feeRecipient, liquidatorProfitBorrowToken);
}

```

## 🔗 Proof of Concept

Say two users setup the same sell limit order.

The first user's order was bought by a user via `BuyCreditMarket`, and the second user's order was bought by `LiquidateWithReplacement`. However, the swap fee is only charged to the first user. This is unfair and is not mentioned anywhere expected in the documents.

## 🔗 Recommended Mitigation Steps

Also charge swap fees during `executeLiquidateWithReplacement`.

aviggiano (Size) acknowledged and commented:

*Feedback from the team:*

- Technically valid observation.
- Whether we fix it or not is a product decision.
- Technically this is a valid one since it is a credit for cash operation, since the chosen borrower sells his credit for cash.
- Just imagine what would happen if that credit sell limit order would have been filled by a credit buy market order: the swap fee would have been charged, and `liquidateWithReplacement()` is essentially not different from doing a credit buy limit order filling a credit sell limit order.
- From a product perspective it is not clear if it was decided to treat this in a special way, i.e., not to charge the swap fee on the credit seller, as it should be, as a form of incentive since it is highly desirable from the protocol perspective to have many such offers as the protocol earns a lot from a liquidation with replacement.
- The `liquidateWithReplacement()` leads to the same outcome of the original loan for the lender and it should be indistinguishable from a standard credit sell limit order being filled for the borrower.
- So regarding the latter, for consistency, the borrower should pay the swap fee; however, it is fair to observe the protocol already makes a big profit from `liquidateWithReplacement()` since it takes the full time value of money that otherwise in case of a standard liquidation would have been earned entirely by

the lender, and since `atm liquidateWithReplacement()` `atm` has some limitations (see the auditors remarks about the problem of running `liquidateWithReplacement()` on large loans, due to the inability of fractionalize debt) then it could make sense to incentivize it.

- I say “could” because that’s a product decision, not relevant for the protocol accounting.

**hansfrieese (judge) commented:**

*Thanks for the detailed feedback. Will keep as a valid Medium.*

## **[M-12] `executeBuyCreditMarket` returns the wrong amount of cash and overestimates the amount that needs to be checked in the variable pool**

Submitted by said, also found by ether\_sky, zhaojohnson, Honour, samuraii77, inzinko, trachev, alix40, hyh, and KupiaSec

When `executeBuyCreditMarket` is called and returns the cash amount, it will return the cash amount without deducting the fee. This results in overestimating the value that needs to be validated by `validateVariablePoolHasEnoughLiquidity`.

### **Proof of Concept**

When `executeBuyCreditMarket` is called, it will transfer `cashAmountIn - fees` to the borrower. This means the amount that needs to be available for withdrawal in the variable pool is `cashAmountIn - fees`.

**<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/BuyCreditMarket.sol#L195>**

```
function executeBuyCreditMarket(State storage state,
BuyCreditMarketParams memory params)
    external
>>>    returns (uint256 cashAmountIn)
    {
        // ...
```

```

uint256 creditAmountOut;
uint256 fees;

if (params.exactAmountIn) {
    cashAmountIn = params.amount;
    (creditAmountOut, fees) = state.getCreditAmountOut({
        cashAmountIn: cashAmountIn,
        maxCashAmountIn: params.creditPositionId ==
RESERVED_ID
        ? cashAmountIn
        : Math.mulDivUp(creditPosition.credit,
PERCENT, PERCENT + ratePerTenor),
        maxCredit: params.creditPositionId == RESERVED_ID
        ? Math.mulDivDown(cashAmountIn, PERCENT +
ratePerTenor, PERCENT)
        : creditPosition.credit,
        ratePerTenor: ratePerTenor,
        tenor: tenor
    });
} else {
    creditAmountOut = params.amount;
    (cashAmountIn, fees) = state.getCashAmountIn({
        creditAmountOut: creditAmountOut,
        maxCredit: params.creditPositionId == RESERVED_ID
? creditAmountOut : creditPosition.credit,
        ratePerTenor: ratePerTenor,
        tenor: tenor
    });
}

// ...

>>> state.data.borrowAToken.transferFrom(msg.sender, borrower,
cashAmountIn - fees);
state.data.borrowAToken.transferFrom(msg.sender,
state.feeConfig.feeRecipient, fees);
}

```

However, `executeBuyCreditMarket` will return `cashAmountIn`, and provide it to `validateVariablePoolHasEnoughLiquidity` and check if the variable pool have `cashAmountIn` amount of token.

<https://github.com/code-423n4/2024-06-size/blob/main/src/Size.sol#L178-L185>

```

function buyCreditMarket(BuyCreditMarketParams calldata
params) external payable override(ISize) whenNotPaused {
    state.validateBuyCreditMarket(params);
>>> uint256 amount = state.executeBuyCreditMarket(params);
    if (params.creditPositionId == RESERVED_ID) {

```

```

state.validateUserIsNotBelowOpeningLimitBorrowCR(params.borrower);
    }
>>>    state.validateVariablePoolHasEnoughLiquidity(amount);
    }

```

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/CapsLibrary.sol#L67-L72>

```

function validateVariablePoolHasEnoughLiquidity(State storage
state, uint256 amount) public view {
    uint256 liquidity =
state.data.underlyingBorrowToken.balanceOf(address(state.data.vari
ablePool));
    if (liquidity < amount) {
        revert
Errors.NOT_ENOUGH_BORROW_ATOKEN_LIQUIDITY(liquidity, amount);
    }
}

```

This will overestimate the amount of liquidity that needs to be available in the variable pool and could cause the call to unnecessarily revert.

## Recommended Mitigation Steps

Update the returned `cashAmountIn` to `cashAmountIn - fees` at the end of `executeBuyCreditMarket`.

```

function executeBuyCreditMarket(State storage state,
BuyCreditMarketParams memory params)
    external
>>>    returns (uint256 cashAmountIn)
    {
        // ...

        uint256 creditAmountOut;
        uint256 fees;

        if (params.exactAmountIn) {
            cashAmountIn = params.amount;
            (creditAmountOut, fees) = state.getCreditAmountOut({
                cashAmountIn: cashAmountIn,
                maxCashAmountIn: params.creditPositionId ==
RESERVED_ID
                    ? cashAmountIn
                    : Math.mulDivUp(creditPosition.credit,
PERCENT, PERCENT + ratePerTenor),
                maxCredit: params.creditPositionId == RESERVED_ID

```

```

        ? Math.mulDivDown(cashAmountIn, PERCENT +
ratePerTenor, PERCENT)
        : creditPosition.credit,
ratePerTenor: ratePerTenor,
tenor: tenor
    });
} else {
    creditAmountOut = params.amount;
    (cashAmountIn, fees) = state.getCashAmountIn({
        creditAmountOut: creditAmountOut,
        maxCredit: params.creditPositionId == RESERVED_ID
? creditAmountOut : creditPosition.credit,
ratePerTenor: ratePerTenor,
tenor: tenor
    });
}

// ...

state.data.borrowAToken.transferFrom(msg.sender, borrower,
cashAmountIn - fees);
state.data.borrowAToken.transferFrom(msg.sender,
state.feeConfig.feeRecipient, fees);
+ cashAmountIn = cashAmountIn - fees;
}

```

## Assessed type

Invalid Validation

aviggiano (Size) confirmed and commented:

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/133>.*

*This is arguably a Low -severity issue since the likelihood is Low*

- Aave being illiquid.
- Order reverting by a fee amount, which is of the order of a percentage point on the dollar.

*Impact is Low (order reverting), but maybe a smaller order could succeed.*

hansfrieze (judge) commented:



*After consideration, keeping it as a valid Medium to remain consistent with [#152](#).*

*Note: For full discussion, see [here](#).*

## 🔗 [M-13] Fragmentation fee is not taken if user compensates with newly created position

Submitted by [SpicyMeatball](#), also found by [OxRobocop](#), [OxAlix2](#), [samuraii77](#), [DanielArmstrong](#), [Rhaydden](#), [ubl4nk](#), [alix40](#), [OxStalin](#), [asui](#), [zanderbyte](#), [carlos\\_alegre](#), [Honour](#), [lanrebayode77](#), [Infect3d](#), [elhaj](#), [Nihavent](#), [Bigsam](#), [trachev](#), [OxRstStn](#), [Jorgect](#), [hyh](#), [serial-coder](#), [KupiaSec](#), [shaflo2](#), [pkqs90](#), [Sentryx](#), [said](#), and [zzebra83](#)

<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Compensate.sol#L119-L125>  
<https://github.com/code-423n4/2024-06-size/blob/main/src/libraries/actions/Compensate.sol#L136>

### 🔗 Impact

User can split his debt credit using the `compensate` function without paying the fragmentation fee.

### 🔗 Proof of Concept

The fragmentation fee normally applies when the initial credit amount is split, creating a new credit position with the fractionalized amount. However, under specific conditions when the `compensate` function is invoked, this fee is not charged.

Consider the following scenario, the user compensates half of his debt ( `amountToCompensate = creditPositionWithDebtToRepay.credit / 2` ) with a new credit position ( `params.creditPositionToCompensateId == RESERVED_ID` ) to the lender:

```
function executeCompensate(State storage state,
CompensateParams calldata params) external {
    emit Events.Compensate(
        params.creditPositionWithDebtToRepayId,
        params.creditPositionToCompensateId, params.amount
    );

    CreditPosition storage creditPositionWithDebtToRepay =
state.getCreditPosition(params.creditPositionWithDebtToRepayId);
```

```

    DebtPosition storage debtPositionToRepay =

state.getDebtPositionByCreditPositionId(params.creditPositionWithD
ebtToRepayId);

>>    uint256 amountToCompensate = Math.min(params.amount,
creditPositionWithDebtToRepay.credit);

    CreditPosition memory creditPositionToCompensate;
    if (params.creditPositionToCompensateId == RESERVED_ID) {
>>        creditPositionToCompensate =
state.createDebtAndCreditPositions({
            lender: msg.sender,
            borrower: msg.sender,
            futureValue: amountToCompensate,
            dueDate: debtPositionToRepay.dueDate
        });

    ---SNIP---

```

In the above code snippet, a new credit position is created with `credit = futureValue = amountToCompensate`. This results in `creditPositionToCompensate.credit` always equating to `amountToCompensate`, leading `exiterCreditRemaining` to consistently be zero:

```

    ---SNIP---

>>    uint256 exiterCreditRemaining =
creditPositionToCompensate.credit - amountToCompensate;

    // credit emission
    state.createCreditPosition({
        exitCreditPositionId:
params.creditPositionToCompensateId == RESERVED_ID
        ? state.data.nextCreditPositionId - 1
        : params.creditPositionToCompensateId,
        lender: creditPositionWithDebtToRepay.lender,
        credit: amountToCompensate
    });
>>    if (exiterCreditRemaining > 0) {
        // charge the fragmentation fee in collateral tokens,
        capped by the user balance
        uint256 fragmentationFeeInCollateral = Math.min(

state.debtTokenAmountToCollateralTokenAmount(state.feeConfig.fragm
entationFee),
            state.data.collateralToken.balanceOf(msg.sender)
        );
        state.data.collateralToken.transferFrom(

```

```
msg.sender, state.feeConfig.feeRecipient,  
fragmentationFeeInCollateral  
);
```

As a result, despite splitting `creditPositionWithDebtToRepay.credit` in half and creating a new credit position, the fragmentation fee is not deducted as expected.

## 🔗 Recommended Mitigation Steps

```
+ uint256 initialCredit = params.creditPositionToCompensateId ==  
RESERVED_ID  
    ? creditPositionWithDebtToRepay.credit  
    : creditPositionToCompensate.credit;  
+ uint256 exiterCreditRemaining = initialCredit -  
amountToCompensate;
```

aviggiano (Size) confirmed and commented:

*This report correctly identifies the root cause and correctly provides a mitigation. This could be the primary issue.*

*Fixed in <https://github.com/SizeCredit/size-solidity/pull/120>.*

## 🔗 Low Risk and Non-Critical Issues

---

For this audit, 11 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by `ether_sky` received the top score from the judge.

*The following wardens also submitted reports: [Oxhacksmithh](#), [alix40](#), [K42](#), [Bauchibred](#), [OxRobocop](#), [Rhaydden](#), [atoko](#), [Shield](#), [NexusAudits](#), and [pkqs90](#).*

### 🔗 [L-01] There should be a grace period for repayment

---

Debt positions have a due date and can be liquidated after this date. Many users will try to repay their debts before the due date. For example, some users may start repaying one day or one hour in advance, depending on their preference. However, there is no 100% guarantee that these repayments will be executed within this period. If the repayment transaction is not included in the block, the debt position might be accidentally liquidated.

At this point, anyone can easily profit by liquidating this debt position. Therefore, it is sensible to set a grace period during which liquidation is not allowed. This can help prevent accidental liquidation and protect users' funds.

This issue can be upgraded as it results in users' funds loss.

## 🔗 [L-02] Lenders can potentially lose funds in the `buyCreditMarket` function

---

In the `buyCreditMarket` function, let's denote some values:

- `A` is the future `credit` value that the `lender` will receive.
- `V` is the `cash` amount that the `lender` should pay to the `borrower`.
- `k` is the `swap fee`.
- `T` is the `tenor`.
- `r` is the `rate`.

According to the formula described in the documentation, when the fragmentation fee is 0, the relationship between these values is as follows:

$$A = V \times (1 + r)$$

$$\text{fee} = V \times k \times T$$

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/AccountingLibrary.sol#L285-L286>

```
function getCreditAmountOut(
    State storage state,
    uint256 cashAmountIn,
    uint256 maxCashAmountIn,
    uint256 maxCredit,
    uint256 ratePerTenor,
    uint256 tenor
) internal view returns (uint256 creditAmountOut, uint256 fees) {
    if (cashAmountIn == maxCashAmountIn) {
        creditAmountOut = maxCredit;
        fees = getSwapFee(state, cashAmountIn, tenor);
    } else if (cashAmountIn < maxCashAmountIn) {
        ...
    } else {
```

```

    revert Errors.NOT_ENOUGH_CREDIT(maxCashAmountIn,
    cashAmountIn);
  }
}

```

There is no guarantee that  $A$  is always larger than the sum of  $V + fee$  due to the absence of a strict relation between  $k$  and  $r$ . Therefore, lenders can potentially pay more funds than what they will receive in the future.

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/BuyCreditMarket.sol#L195-L196>

```

function executeBuyCreditMarket(State storage state,
BuyCreditMarketParams memory params)
    external
    returns (uint256 cashAmountIn)
{
    state.data.borrowAToken.transferFrom(msg.sender, borrower,
cashAmountIn - fees);
    state.data.borrowAToken.transferFrom(msg.sender,
state.feeConfig.feeRecipient, fees);
}

```

The loss can increase when there is a fragmentation fee. This issue can be upgraded as it can lead to a user's funds loss.

🔗 **[L-03] Users who have a loan offer should have some USDC to create a loan .**

In the `sellCreditMarket` function, the transaction will be reverted if the lender does not have enough USDC.

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/SellCreditMarket.sol#L201-L202>

```

function executeSellCreditMarket(State storage state,
SellCreditMarketParams calldata params)
    external
    returns (uint256 cashAmountOut)
{
    state.data.borrowAToken.transferFrom(params.lender,
msg.sender, cashAmountOut);
}

```

```
state.data.borrowAToken.transferFrom(params.lender,
state.feeConfig.feeRecipient, fees);
}
```

To prevent this, we can add a check for the lender's balance in the pre-check stage.

🔗 **[L-04] The `isMulticall` flag is not correctly reset to `false` in the `multicall` function.**

---

In the `multicall` function, we set `isMulticall` to true at the beginning and reset it to false at the end.

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/Multicall.sol#L44>

```
function multicall(State storage state, bytes[] calldata data)
internal returns (bytes[] memory results) {
    state.data.isMulticall = true;

    ...

    state.data.isMulticall = false;
}
```

However, if another `multicall` function is called within the initial `multicall` function, the `isMulticall` flag will be reset to false when the inner `multicall` function execution ends. If a subsequent `deposit` function follows, the cap check will be performed, which violates the `multicall` function design.

To prevent this, we should not allow an inner `multicall` function within another `multicall` function.

```
function multicall(State storage state, bytes[] calldata data)
internal returns (bytes[] memory results) {
+   if (state.data.isMulticall == true) revert();
    state.data.isMulticall = true;

    ...

    state.data.isMulticall = false;
}
```

## [L-05] Users cannot make compensation when the due date is within the minimum tenor period

---

*Note: At the judge's request [here](#), this downgraded issue from the same warden has been included in this report for completeness.*

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/actions/Compensate.sol#L60-L62>

### Impact

Users can repay their debts partially. If they have other credits with due dates earlier than the debt they want to compensate for, they can always make the compensation regardless of whether the due date falls within the minimum tenor period.

However, it is not possible to create new debt/credit and compensate using this new credit if the due date is within the minimum tenor period. In such cases, the newly created debt and credit positions are just for compensation, and the owner is the same user, so the minimum tenor period check should not be necessary.

Imagine the minimum tenor period is long and a user has a large debt position. If the ETH price drops, their position may soon become liquidatable. If the user has some USDC but no ETH, they would want to repay their debt partially to increase their CR again. However, if the due date is within the minimum tenor period, they cannot do this. As a result, their position can be liquidated, leading to a loss.

The size of the loss depends on the debt size, so users will try to minimize the loss by reducing the debt size. Nevertheless, due to the unnecessary minimum tenor period check, users can still experience losses.

### Proof of Concept

Imagine Bob borrowed USDC from Alice, and Candy borrowed USDC from Bob. Both debts have the same due dates and are within the minimum tenor period. There is a fluctuation in the ETH price, prompting both users to want to reduce their debts.

In this scenario, Bob can compensate using the credit that Candy will pay to him. However, Candy cannot reduce his debt due to the minimum tenor period check.

```
function validateCompensate(State storage state, CompensateParams
calldata params) external view {
    if (params.creditPositionToCompensateId == RESERVED_ID) {
        uint256 tenor = debtPositionToRepay.dueDate -
        block.timestamp;
```

```

        // validate tenor
        if (tenor < state.riskConfig.minTenor || tenor >
state.riskConfig.maxTenor) {
            revert Errors.TENOR_OUT_OF_RANGE(tenor,
state.riskConfig.minTenor, state.riskConfig.maxTenor); // @audit,
here
        }
    }
}

```

As previously mentioned, these newly created debt and credit positions are solely for compensation, and Candy owns both positions. This does not constitute actual lending/borrowing. Therefore, users should be allowed to create these positions even if the due date is within the minimum tenor period. It should be permissible because compensation can occur using credits with due dates within the minimum tenor period. In this context, there should be no distinction between these two methods.

Please add below test to the `test/local/actions/Compensate.t.sol` :

```

function test_Compensate_can_not_compensate_in_min_tenor() public
{
    uint256 minTenor = size.riskConfig().minTenor;

    _deposit(alice, usdc, 100e6);
    _deposit(bob, usdc, 50e6);
    _deposit(candy, usdc, 50e6);

    _deposit(bob, weth, 100e18);
    _deposit(candy, weth, 100e18);

    _buyCreditLimit(alice, block.timestamp + 365 days,
YieldCurveHelper.pointCurve(365 days, 0.03e18));
    _buyCreditLimit(bob, block.timestamp + 365 days,
YieldCurveHelper.pointCurve(365 days, 0.03e18));

    uint256 debtPositionIdOfBob = _sellCreditMarket(bob, alice,
RESERVED_ID, 100e6, 365 days, true);
    uint256 creditPositionIdOfBob =
size.data().nextCreditPositionId - 1;

    uint256 debtPositionIdOfCandy = _sellCreditMarket(candy, bob,
RESERVED_ID, 50e6, 365 days, true);
    uint256 creditPositionIdOfCandy =
size.data().nextCreditPositionId - 1;

    vm.warp(block.timestamp + 365 days - minTenor + 10);

    vm.prank(candy);
    // Candy cannot repay his debt

```



```

        vm.expectRevert(
            abi.encodeWithSelector(
                Errors.TENOR_OUT_OF_RANGE.selector, 3590, minTenor,
size.riskConfig().maxTenor
            )
        );
        size.compensate(
            CompensateParams({
                creditPositionWithDebtToRepayId:
creditPositionIdOfCandy,
                creditPositionToCompensateId: RESERVED_ID,
                amount: 20e6
            })
        );

        vm.prank(bob);
        // Bob can repay his debt using his another credit
        size.compensate(
            CompensateParams({
                creditPositionWithDebtToRepayId:
creditPositionIdOfBob,
                creditPositionToCompensateId: creditPositionIdOfCandy,
                amount: 20e6
            })
        );
    }
}

```

## 🔗 Recommended Mitigation Steps

Please remove the minimum tenor period check from the compensate function.

## 🔗 Assessed type

Error

aviggiano (Size) disputed and commented:

*It is a matter of consistency. Initially, that check was placed to avoid active users with loans close to due date to “dump” them in compensating their debt to avoid the risk of them getting overdue.*

*There is no difference between a newly minted credit and an existing credit in terms of the risk of them becoming overdue since also the newly minted credit can become overdue if the minter does not repay on time.*

*In fact, imagine there are 3 subjects: A,B,C and both B and C have a debt with A.*

*In the initial example, we have that B mints a credit with  $\text{dueDate} - \text{block.timestamp} = \text{minTenor} - \text{epsilon}$  and he would like to compensate. But at the moment he can't because of that check. However, let's assume the check is removed and A receives a credit maturing in  $\text{minTenor} - \text{epsilon}$ .*

*Now let's assume that B mints the credit at  $\text{dueDate} - \text{block.timestamp} = \text{minTenor} + \text{epsilon}$  so that it can be transferred (by the way, what's the current implementation for the transfer eligibility criteria? Is it  $\text{dueDate} - \text{block.timestamp} > 0$  so just not overdue or  $\text{dueDate} - \text{block.timestamp} > \text{minTenor}$  ?) to C who buys it.*

*Then after  $2 * \text{epsilon}$  time we have that C would like to compensate his debt to A but the compensation gets prevented since now we have  $\text{dueDate} - \text{block.timestamp} = \text{minTenor} - \text{epsilon}$ . So relaxing that check in the first case A receives a credit maturing at  $\text{dueDate} - \text{block.timestamp} = \text{minTenor} - \text{epsilon}$  since it is considered safe while in the second case A can't receive a credit maturing at  $\text{dueDate} - \text{block.timestamp} = \text{minTenor} - \text{epsilon}$  since it is considered unsafe.*

*Here is the inconsistency.*

*So either they are both allowed or both not allowed. In the current implementation, they are both not allowed and it is consistent.*

#### lanrebayode77 (warden) commented:

*I still think this report is a valid issue; Imagine an instance with Alice who is a regular honest user of Size. Alice currently has a debt position of 1,000,000 adequately collateralised with 1,300,000 worth of collateral token.*

*Alice faces real issue where only 900,000 debt token is available at the moment to pay back. Alice is willing to pay back timely but currently short of funds, but has collateral that cannot be withdrawn until debt position decreases.*

*Since, debt is almost due with no other option, Alice tries to compensate her loan by splitting it to create a new debt and credit position worth 900,000, so `repay()` can be called with all the available debt tokens, reduce current `debtPosition` and make some collateral available for withdrawal so it can be converted to debt tokens which will then be used to pay up the remaining debt.*

*The user flow described above is a honest one, with nothing broken (min credit is maintained and Lender funds are available on or before due date).*

*Note according to the comment above, no risk of extending overdue date since both loans will be due for liquidation anyway if payment is not made.*

*There is no difference between a newly minted credit and an existing credit in terms of the risk of them becoming overdue since also the newly minted credit can become overdue if the minter does not repay on time.*

*Also, the only due date validation done when another credit is used to compensate is to ensure that the credit being used has a earlier due date and does not include `mitTenor` check*

```
if (
    debtPositionToRepay.dueDate
    <
    state.getDebtPositionByCreditPositionId(params.creditPositionToCom
pensateId).dueDate
) {
    revert Errors.DUE_DATE_NOT_COMPATIBLE(
        params.creditPositionWithDebtToRepayId,
        params.creditPositionToCompensateId
    );
}
```

*I believe there is no need to restrict compensation of loan with new credit when duration is less than `minTenor` , since the function serve to increase payment flexibility for users:*

1. Lenders get funds when payment is made asap.
2. If payment is not made asap, liquidation is inevitable and overdue fees are still incurred by borrower.

**hansfrieze (judge) decreased severity to Low and commented:**

*I agree that validating the minimum tenor for the same borrower is unnecessary, and the suggested mitigation is reasonable. However, this issue pertains to the protocol's*

design and is insufficient to be classified as a Medium. I will downgrade this to a valid QA.

## [L-06] When the `borrowAToken` cap is reached, users may not always be able to repay their debt in some cases due to precision errors

*Note: At the judge's request [here](#), this downgraded issue from the same warden has been included in this report for completeness.*

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/token/NonTransferrableScaledToken.sol#L98-L100>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/DepositTokenLibrary.sol#L60>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/libraries/logic/SupplyLogic.sol#L69-L74>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/libraries/logic/SupplyLogic.sol#L69-L74>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/tokenization/base/ScaledBalanceTokenBase.sol#L72>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/tokenization/base/ScaledBalanceTokenBase.sol#L72>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/libraries/math/WadRayMath.sol#L83-L92>

<https://github.com/aave/aave-v3-core/blob/b74526a7bc67a3a117a1963fc871b3eb8cea8435/contracts/protocol/libraries/math/WadRayMath.sol#L83-L92>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/CapsLibrary.sol#L35-L40>

<https://github.com/code-423n4/2024-06-size/blob/8850e25fb088898e9cf86f9be1c401ad155bea86/src/libraries/CapsLibrary.sol#L35-L40>

## Impact

Users should always be able to repay their debts, even if the `borrowAToken` cap has been reached. This is crucial because positions can be liquidated if debts aren't repaid on time, leading to significant losses for users. However, in some cases, users cannot repay their debts due to precision errors. This issue stems from Size and Aave using different mathematical formulas in the calculation of scaled balances.

## Proof of Concept

Imagine the `borrowAToken` cap has been reached, and the current scaled total supply is denoted as  $T$ . To calculate the total supply, we use the formula  $T * L / R$ , where  $L$  represents the current liquidity index of the Aave pool and  $R$  is set as `RAY` ( $1e27$ ).

```
function totalSupply() public view override returns (uint256) {
    return _unscale(scaledTotalSupply());
}
function _unscale(uint256 scaledAmount) internal view returns
(uint256) {
    return Math.mulDivDown(scaledAmount, liquidityIndex(),
WadRayMath.RAY);
}
```

Consider a user who has a debt  $D$  and wishes to repay it before maturity. This process should be achievable using multicall transactions, which typically involves two steps:

1. Deposit  $D$  USDC into Size.
2. These  $D$  USDC are supplied into the Aave Pool.

```
function depositUnderlyingBorrowTokenToVariablePool(State storage
state, address from, address to, uint256 amount)
    external
{
    uint256 scaledBalanceBefore =
aToken.scaledBalanceOf(address(this));

state.data.variablePool.supply(address(state.data.underlyingBorrow
Token), amount, address(this), 0); // @audit, here
}
```

In the Aave pool, the scaled balance of these  $D$  USDC is minted back into Size.

```
function executeSupply(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    DataTypes.UserConfigurationMap storage userConfig,
    DataTypes.ExecuteSupplyParams memory params
) external {
    bool isFirstSupply = IAToken(reserveCache.aTokenAddress).mint(
        msg.sender,
        params.onBehalfOf,
        params.amount, // @audit, here
        reserveCache.nextLiquidityIndex
```

```
);
}
```

Aave employs a slightly different mathematical approach to calculate scaled balances.

```
function _mintScaled(
    address caller,
    address onBehalfOf,
    uint256 amount,
    uint256 index
) internal returns (bool) {
    uint256 amountScaled = amount.rayDiv(index); // @audit, here
    require(amountScaled != 0, Errors.INVALID_MINT_AMOUNT);
}
```

The scaled balance of  $D$  USDC is given by  $(D * R + L / 2) / L$ .

```
function rayDiv(uint256 a, uint256 b) internal pure returns
(uint256 c) {
    assembly {
        if or(iszero(b), iszero(iszero(gt(a, div(sub(not(0), div(b,
2)), RAY))))) {
            revert(0, 0)
        }

        c := div(add(mul(a, RAY), div(b, 2)), b) // @audit, here
    }
}
```

Therefore, the increase in `borrowAToken` supply will be determined by this value.

```
function depositUnderlyingBorrowTokenToVariablePool(State storage
state, address from, address to, uint256 amount)
    external
{
    uint256 scaledBalanceBefore =
aToken.scaledBalanceOf(address(this));

    state.data.variablePool.supply(address(state.data.underlyingBorrow
Token), amount, address(this), 0);
    uint256 scaledAmount = aToken.scaledBalanceOf(address(this)) -
scaledBalanceBefore;

    state.data.borrowAToken.mintScaled(to, scaledAmount); //
@audit, here
}
```

Afterward, the user attempts to repay his debt. Obviously the debt decrease is  $D$ .

In certain cases,  $(D * R + L / 2) / L$  can exceed  $D$  by just 1 wei. For instance, if  $D$  is 103,517,589 (can occur when  $D$  is an odd number) and  $L$  is  $2e27$ . This is possible and can easily occur with different values of  $L$ , then  $(D * R + L / 2) / L$  becomes 103,517,590. Please check these values in the below test.

As a result of this precision discrepancy, the validation check will fail, preventing the user from repaying their debt as intended.

```
function validateBorrowATokenIncreaseLteDebtTokenDecrease(
    State storage state,
    uint256 borrowATokenSupplyBefore,
    uint256 debtTokenSupplyBefore,
    uint256 borrowATokenSupplyAfter,
    uint256 debtTokenSupplyAfter
) external view {
    if (borrowATokenSupplyAfter >
state.riskConfig.borrowATokenCap) {
        uint256 borrowATokenSupplyIncrease =
borrowATokenSupplyAfter > borrowATokenSupplyBefore
        ? borrowATokenSupplyAfter - borrowATokenSupplyBefore
        : 0;
        uint256 debtATokenSupplyDecrease =
debtTokenSupplyBefore > debtTokenSupplyAfter ?
debtTokenSupplyBefore - debtTokenSupplyAfter : 0;

        if (borrowATokenSupplyIncrease > debtATokenSupplyDecrease)
{ // @audit, here
            revert
Errors.BORROW_ATOKEN_INCREASE_EXCEEDS_DEBT_TOKEN_DECREASE(
                borrowATokenSupplyIncrease,
debtATokenSupplyDecrease
            );
        }
    }
}
```

Please add below test to the `test/local/actions/Multicall.t.sol`:

To conduct this test, we should implement the suggestion from that report, which involves using the total supply instead of the balance of `Size` in the multicall cap check.

```
function test_Multicall_repay_when_borrowAToken_cap() public {
    NonTransferrableScaledToken borrowAToken =
size.data().borrowAToken;
    IERC20Metadata debtToken =
IERC20Metadata(address(size.data().debtToken));
```

```

    _setLiquidityIndex(2e27);
    uint256 cap = 1000e6;
    _updateConfig("borrowATokenCap", cap);

    uint256 tenor = 365 days;
    _deposit(alice, usdc, cap);
    _buyCreditLimit(alice, block.timestamp + 365 days,
YieldCurveHelper.pointCurve(tenor, 0.03e18));

    _deposit(bob, weth, 100e18);
    uint256 amount = 100e6 + 1;
    uint256 debtPositionId = _sellCreditMarket(bob, alice,
RESERVED_ID, amount, tenor, false);

    // 103517589 (bob's debt)
    uint256 debtAmount = debtToken.balanceOf(bob);
    _mint(address(usdc), bob, debtAmount);
    _approve(bob, address(usdc), address(size), debtAmount);

    // Bob should always be able to repay his full debt
    bytes[] memory data = new bytes[](2);
    data[0] = abi.encodeCall(size.deposit, DepositParams({token:
address(usdc), amount: debtAmount, to: bob}));
    data[1] = abi.encodeCall(size.repay,
RepayParams({debtPositionId: debtPositionId}));

    // The increase in the borrowAToken total supply exceeds the
debt decrease by just 1 we
    vm.expectRevert(abi.encodeWithSelector(Errors.BORROW_ATOKEN_INCREA
SE_EXCEEDS_DEBT_TOKEN_DECREASE.selector, debtAmount + 1,
debtAmount));
    vm.prank(bob);
    size.multicall(data);
}

```

## Recommended Mitigation Steps

To ensure consistency, aligning Size with Aave's calculation methods would be beneficial, though potential changes require careful consideration. Alternatively, a simple solution could be to incorporate a tolerance of 1 wei in the validation check.

## Assessed type

Math

[aviggiano \(Size\) disputed and commented:](#)



*The usage of `mulDivDown` instead of `rayDiv` was chosen to mitigate Issue 6.3 from the security review [here](#).*

*Additionally, note how `mulDivDown` is safer for the protocol since it discredits the user by at most 1 wei, which is different from what the Aave deposit would give them. In contrast, using `rayDiv` could potentially give them 1 wei more in, for example, `claim`.*

**[hansfrieze \(judge\)](#) decreased severity to Low and commented:**

*After consideration, I believe QA is more appropriate given the unlikely scenario and the existing resolution for borrowers.*

*Note: For full discussion, see [here](#).*

## Disclosures

---

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top

[Twitter](#) | [Discord](#) | [GitHub](#) | [Blog](#) | [Newsletter](#) | [Media kit](#) | [Careers](#)