Size v1 Security Review

Auditors

Hyh, Lead Security Researcher

0xLeastwood, Lead Security Researcher

Slowfi, Security Researcher

0x4non, Associate Security Researcher

Report prepared by: Lucas Goiriz

May 30, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Size is a marketplace for fixed rate loans of any size.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of size-v2-solidity according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Size engaged with Spearbit to review the size-v2-solidity protocol. In this period of time a total of **20** issues were found.

Summary

Project Name	Size
Repository	size-v2-solidity
Commit	54b9da...949794
Type of Project	DeFi, Lending
Audit Timeline	Apr 15 to May 3

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	0	0
High Risk	2	0	0
Medium Risk	4	0	0
Low Risk	2	0	0
Gas Optimizations	1	0	0
Informational	9	0	0
Total	**20**	**0**	**0**

5 Findings

5.1 Critical Risk

5.1.1 Paying `repayFee` can be avoided by any borrower via `BorrowerExit` or `BorrowAsMarketOrder`

Severity: Critical Risk

Context: BorrowerExit.sol#L95-L96, BorrowAsMarketOrder.sol#L166-L180

Description: A borrower can remove their `repayFee`, rendering it to a dust amount, with the help of two accounts by:

1. `BorrowerExit` The borrower (`Alice.1`) right after new borrowing was made can substitute full `repayFee` with `earlyBorrowerExitFee` by atomically setting up another collateralized account `Alice.2` and running `Alice.2.BorrowAsLimitOrder(huge APR) -> Alice.1.BorrowerExit(to Alice.2) -> Alice.2.BorrowAsLimitOrder(null) -> Alice.1.Withdraw(all).` I.e. `issuanceValue` can be manipulated to become dust on borrower exiting to an own account, which renders `repayFee` meaningless. Also, `Alice.2` collateral requirements will be lower compared to the initial debt position since `repayFee` part will be close to zero.
2. `BorrowAsMarketOrder`
   - `Bob.1` is a lender posted some dust collateral (say `10 USDC` valued), creates a lender offer with a significant APR
   - `Bob.2` borrows from `Bob.1` with their target `faceValue` and `dueDate` (e.g., `faceValue = 1e6 USDC` maturing in `1 year`), while `issuanceValue` and `repayFee` are close to zero due to APR based calculation
   - `Bob.1` borrows from market for given `faceValue` and `dueDate` using `Bob.2` debt linked credit position as receivables
   - This way Bob go `_borrowGeneratingDebt()` route interacting with their own account, and then `_borrowFromCredit()` route interacting with the market lending offer
   - At `dueDate` account `Bob.2` fully repays `faceValue`
   - Bob effectively borrowed `faceValue` due at `dueDate` paying no material fees to the protocol
   - The collateral requirements will be similarly lighter due to `repayFee` part removal

Both paths provide an attacking borrower with the intended debt position that has no `repayFee` component. There are no material prerequisites, it can be routinely done by any borrower instead of taking from the market directly.

Likelihood: High (no low probability preconditions) + Impact: High (repay fee is the major revenue source for the protocol) = Severity: Critical.

Recommendation: The reconfiguration of the fee mechanics, namely moving to the credit swap origination fees discussed, should disable the surface.

The simplest approach for the current design is basing `repayFee` on the `faceValue` instead of the `issuanceValue` (having long term borrowers effectively paying higher share of the fees is the cost here).

Size: Fixed in commit 420b4061.

Spearbit: Acknowledged but not reviewed.

5.1.2 Reverse market is not supported due to improper escalations

Severity: Critical Risk

Context: AccountingLibrary.sol#L26-L35, RiskLibrary.sol#L29-L40

Description: The current development of Size protocol is designed to operate with USDC as borrow token and WETH as collateral. Size team also intends to deploy a reverse market version of the protocol with WETH as borrow token and USDC as collateral.

However the current implementation does not support the reverse market. The lack of deployment configuration values for this version introduces different potential issues based on how this values are escalated.

The main identified function prone to fail is `debtTokenAmountToCollateralTokenAmount`. This function escalated the `debtTokenAmount` to Wad, so 18 decimals. Then multiplies it by 1e18 and divides it by the price returned by the price feed, escalated to 18 decimals. However on a reverse market, this value would be as well returned escalated to 18 decimals but representing a USDC amount. Effectively breaking any further calculation or transfer based on this returned amount.

Also, if certain configuration values are not escalated differently other functions can be affected. For example, if `crOpening` is not changed to 6 decimals on reverse market, the function `collateralRatio` may return values escalated to 6 decimals and thus reverting when validating if a user is below the limit borrow collateral ration. Moreover if the `crLiquidation` is not escalated as well, the function `isUserUnderwater` may compare 6 decimal values with 18 decimal values.

Recommendation: It is strongly advised to first identify how the deployment of the reverse market want to be performed. Based on that look to the previous identified issues and correct them if applicable. Finally it is also advised to test the reverse market as throughly as the current version has been tested to avoid potential escalation issues.

Size: Fixed in commit 131d3a93. We have decided to remove support for the reverse market in this version of the protocol. The only supported pair will be (W)ETH/USDC as Collateral/Borrow token.

Spearbit: Acknowledged but not reviewed.

5.2 High Risk

5.2.1 Early self liquidations receive a portion of future fees to be paid by other creditors

Severity: High Risk

Context: SelfLiquidate.sol#L60-L64

Description: Self liquidations are permitted when a creditor is at loss. Put simply, this means when the total debt of a position exceeds it's share of collateral then creditors may self liquidate to realize the loss instead of being further exposed to the borrower. The flow of self liquidation looks like the following:

- Liquidation eligibility is checked on the debt position.
- Repay fees are paid pro-rata by the borrower in proportion to the creditor's contribution of `debtPosition.faceValue`.
- `assignedCollateral` is calculated based off of `creditPosition` in a pro-rata fashion.

`assignedCollateral` utilizes the debt position's total collateral and does not consider fees that other creditors will pay when they perform their respective liquidations. In effect, the first creditor has a percentage claim on all collateral which includes unpaid fees.

Recommendation: The implementation of `LoanLibrary.getCreditPositionProRataAssignedCollateral()` can be modified such that `repayFee` is deducted from the debt position's collateral amount is adjusted for unpaid fees.

Size: Fixed in commit 420b4061.

Spearbit: Acknowledged but not reviewed.

5.2.2 Validation of `APR` in `BuyMarketCredit` leads to underestimation of borrower requirements

Severity: High Risk

Context: BuyMarketCredit.sol#L61-L65

Description: The implementation of the `BuyMarketCredit` checks if the `APR` exceeds `maxAPR` to decide whether to proceed with a lending operation. However, the intention, as deduced from the business logic, seems to focus on ensuring that the `APR` should not fall below a certain threshold, which is beneficial for the lender in terms of returns. The current implementation may lead to transactions that do not satisfy the lender's intended minimum yield.

Recommendation: Consider revising the conditional check and error message to ensure that the `APR` does not fall below a defined minimum acceptable rate (`minAPR`).

```
--- a/src/libraries/fixed/actions/BuyMarketCredit.sol
+++ b/src/libraries/fixed/actions/BuyMarketCredit.sol
@@ -60,8 +60,8 @@ library BuyMarketCredit {

         // validate maxAPR
         uint256 apr = borrowOffer.getAPRByDueDate(state.oracle.variablePoolBorrowRateFeed,
↪  debtPosition.dueDate);
-        if (apr > params.maxAPR) {
-            revert Errors.APR_GREATER_THAN_MAX_APR(apr, params.maxAPR);
+        if (apr < params.minAPR) {
+            revert Errors.APR_LOWER_THAN_MIN_APR(apr, params.minAPR);
         }
```

Size: Fixed in commit 0f759ac8.

Spearbit: Acknowledged but not reviewed.

5.3 Medium Risk

5.3.1 Self liquidations are profitable under certain collateralization ratios

Severity: Medium Risk

Context: SelfLiquidate.sol#L31-L44

Description: The specific eligibility for a self-liquidation as per the spec is when `debtInCollateralToken` exceeds the total collateral allocated to the credit position. Assuming there is only *one* lender, it seems possible that a creditor could self-liquidate and profit from liquidation because `debtInCollateralToken` considers the debt position's total debt and not just `debtPosition.faceValue`.

```
function validateSelfLiquidate(State storage state, SelfLiquidateParams calldata params) external view {
    CreditPosition storage creditPosition = state.getCreditPosition(params.creditPositionId);
    DebtPosition storage debtPosition =
↪   state.getDebtPositionByCreditPositionId(params.creditPositionId);

    uint256 assignedCollateral = state.getCreditPositionProRataAssignedCollateral(creditPosition);
    uint256 debtInCollateralToken =
↪   state.debtTokenAmountToCollateralTokenAmount(debtPosition.getTotalDebt());

    // validate creditPositionId
    if (!state.isCreditPositionSelfLiquidatable(params.creditPositionId)) {
        revert Errors.LOAN_NOT_SELF_LIQUIDATABLE(
            params.creditPositionId,
            state.collateralRatio(debtPosition.borrower),
            state.getLoanStatus(params.creditPositionId)
        );
    }
    if (!(assignedCollateral < debtInCollateralToken)) {
        revert Errors.LIQUIDATION_NOT_AT_LOSS(params.creditPositionId, assignedCollateral,
↪   debtInCollateralToken);
    }

    // validate msg.sender
    if (msg.sender != creditPosition.lender) {
        revert Errors.LIQUIDATOR_IS_NOT_LENDER(msg.sender, creditPosition.lender);
    }
}
```

In practice, self-liquidations are profitable for creditors when total debt is within the range of (assuming there is only a single creditor):

```
state.debtTokenAmountToCollateralTokenAmount(creditPosition.credit) < assignedCollateral <
↪   debtInCollateralToken
```

Creditors profit on the collateral that is required to back the debt position's `repayFee` and `overdueLiquidatorReward`.

The total profit for creditors is strictly limited to:

```
0 < liquidationProfit < state.debtTokenAmountToCollateralTokenAmount(debtPosition.repayFee +
↪   debtPosition.overdueLiquidatorReward)
```

Recommendation: Consider updating the parameter for calculating `debtInCollateralToken` via `state.debtTokenAmountToCollateralTokenAmount(debtPosition.getTotalDebt())` to be `debtPosition.faceValue` instead. This ensures that a self-liquidation is only ever eligible when the position is undercollateralized. In all other cases, standard liquidations should be incentivized.

Size: Fixed:

1. Commit 420b4061 replaces `repayFee` by `swapFee`, which is charged when a loan is generated.
2. Commit bfdabea4 removes `overdueLiquidatorReward`.

Spearbit: Acknowledged but not reviewed.

5.3.2 `LiquidateWithReplacement` might not be available for the big enough debt positions

Severity: Medium Risk

Context: LiquidateWithReplacement.sol#L113-L122, Size.sol#L228-L238

Description: `liquidatorProfitBorrowAsset` can be substantial for the biggest debt positions, but it is assumed that the corresponding big enough `params.borrower` always be found, i.e. that there exists one position both having an eligible offer and enough free collateral to cover the whole debt position being liquidated at once:

- Size.sol#L228-L238

```
    function liquidateWithReplacement(LiquidateWithReplacementParams calldata params)
        // ...
    {
        state.validateLiquidateWithReplacement(params);
        (liquidatorProfitCollateralAsset, liquidatorProfitBorrowAsset) =
↪   state.executeLiquidateWithReplacement(params);
      state.validateUserIsNotBelowOpeningLimitBorrowCR(params.borrower); // <<
```

This might not be the case for prolonged periods of time, while creating such position in a flash loan manner is not possible (since the new debt position has to stay), so a usual `Liquidate -> Withdraw` might be eventually performed instead, which is compatible with flash borrowing of the face value and selling the collateral proceeds with repaying the loan atomically, i.e. there are no additional position existence requirements there.

This way protocol will deprive itself from `liquidatorProfitBorrowAsset` in some cases when it's the biggest.

Recommendation: Consider introducing the possibility to supply a number of borrowers who can cover the liquidated debt position cumulatively (e.g., similarly to treating receivables in `BorrowAsMarketOrder`, BorrowAsMarketOrder.sol#L124-L152).

Size: Acknowledged. This was a known limitation, and since `liquidateWithReplacement` is not vital, we will revise this feature in a future version of the protocol.

Spearbit: Acknowledged but not reviewed.

5.3.3 Unexpected fee deduction on self-liquidations

Severity: Medium Risk

Context: SelfLiquidate.sol#L52-L73

Description: The `selfLiquidate` function is designed to allow lenders to liquidate their positions without external liquidator involvement, improving protocol health by ensuring timely collateral recovery without profitability for the liquidator. According to the technical documentation 3.5.3.2 Self Liquidation, no fees should be charged during this process. However, the current implementation incorrectly charges a repay fee before computing the collateral to be assigned to the lender, thereby reducing the collateral amount available contrary to the documentation.

Recommendation: Amend the `executeSelfLiquidate` function to omit the charging of fees during the self-liquidation process:

```
--- a/src/libraries/fixed/actions/SelfLiquidate.sol
+++ b/src/libraries/fixed/actions/SelfLiquidate.sol
@@ -57,7 +57,7 @@ library SelfLiquidate {

         uint256 credit = creditPosition.credit;

-        uint256 repayFeeProRata = state.chargeRepayFeeInCollateral(debtPosition, credit);
+        uint256 repayFeeProRata = Math.mulDivDown(debtPosition.repayFee, credit,
↪   debtPosition.faceValue);
         uint256 assignedCollateral = state.getCreditPositionProRataAssignedCollateral(creditPosition);
         (uint256 debtProRata, bool isFullRepayment) = debtPosition.getDebtProRata(credit,
↪   repayFeeProRata);
         state.data.debtToken.burn(debtPosition.borrower, debtProRata);
```

Size: Fixed in commit a2bafef2.

Spearbit: Acknowledged but not reviewed.

5.3.4 Maximum capital checks may prevent deposits

Severity: Medium Risk

Context: Size.sol#L130-L131

Description: The `deposit` functions performs two validation checks after depositing either the collateral or borrow token. These two validations attempt to ensure that the maximum amounts defined by the Size team of each token on the protocol are not exceeded. The functions that perform the checks are:

- `validateCollateralTokenCap`
- `validateBorrowATokenCap`

The maximum amounts are initially designed to avoid an excessive amount of funds entering the protocol. Nevertheless there are certain scenarios where these checks may arise functional problems.

The exposed scenario for illustrating this issue starts with a user attempting to deposit ETH as collateral. But the amount of borrow token, calculated as the total balance of `aToken` from funds deposited on Aave has increased over the maximum amount, the transaction reverts.

This can also have a higher impact when depositing collateral to avoid liquidation, as identified by Size team.

Recommendation: After a discussion with Size team, the final conclusion for fixing this issue and any other related with maximum amounts check is to just control the amount of emitted debt token.

Size: Fixed in commit fee6ab85.

Spearbit: Acknowledged but not reviewed.

5.4 Low Risk

5.4.1 Market order borrows may revert due to missing edge case

Severity: Low Risk

Context: BorrowAsMarketOrder.sol#L107-L153

Description: When a borrower attempts to execute a market order against a specific lender, the borrower can opt to borrow from their existing credit positions that will expire before the due date of the new debt position being created. This maintains the availability of credit redemption when each debt position expires. Credit is transferred from `msg.sender` to `params.lender` who effectively buys the borrower's credit.

To ensure the correct leftover amount is used to generate debt, `_borrowFromCredit()` will skip credit positions with `deltaAmountIn < state.riskConfig.minimumCreditBorrowAToken`. However, this fails to consider when the last credit position pushes `exitCreditPositionId` below `state.riskConfig.minimumCreditBorrowAToken`. Meaning the borrower will be unable to fully execute their borrow market order as it will revert before being able to generate the rest of `amountOutLeft` as debt. This revert happens explicitly within the `AccountingLibrary.createCreditPosition()` function.

Recommendation: Modify the `_borrowFromCredit()` function to skip when the following conditions are satisfied.

```
uint256 exitCreditAmount = creditPosition.credit - deltaAmountIn;
if (deltaAmountIn < state.riskConfig.minimumCreditBorrowAToken || (exitCreditAmount != 0 &&
↪  exitCreditAmount < state.riskConfig.minimumCreditBorrowAToken)) {
    continue;
}
```

Size: Fixed in commit 420b4061.

Spearbit: Acknowledged but not reviewed.

5.4.2 `faceValue` precision can be lowered by additional `mulDivDown` in `LendAsMarketOrder`

Severity: Low Risk

Context: LendAsMarketOrder.sol#L79-L86

Description: When `!params.exactAmountIn` the `mulDivDown` operation is done twice in `LendAsMarketOrder`, introducing additional rounding errors to the `faceValue` value.

Recommendation: Consider unifying the calculations similarly to BuyMarketCredit.sol#L82-L88, e.g.:

- LendAsMarketOrder.sol#L79-L86

```
    uint256 ratePerMaturity =
        borrowOffer.getRatePerMaturityByDueDate(state.oracle.variablePoolBorrowRateFeed,
↪   params.dueDate);
  + uint256 faceValue;
    if (params.exactAmountIn) {
        issuanceValue = params.amount;
  +     faceValue = Math.mulDivDown(params.amount, PERCENT + ratePerMaturity, PERCENT);
    } else {
        issuanceValue = Math.mulDivUp(params.amount, PERCENT, PERCENT + ratePerMaturity);
  +     faceValue = params.amount;
    }
  - uint256 faceValue = Math.mulDivDown(issuanceValue, PERCENT + ratePerMaturity, PERCENT);
```

Size: Fixed in commit 3d70e46b.

Spearbit: Acknowledged but not reviewed.

5.5 Gas Optimization

5.5.1 Avoid `forceApprove` for self transfers on ETH deposits

Severity: Gas Optimization

Context: Deposit.sol#L63

Description: The current implementation calls the `forceApprove` function from the WETH contract to perform a transferFrom to address(this). This two function calls are not required and can be gas optimised.

Recommendation: Consider erasing the forceApprove and placing the next check before L17 of CollateralLibrary:

```
if (from != address(this))
    underlyingCollateralToken.transferFrom(from, address(this), amount);
```

Size: Acknowledged. Won't Fix. Although the revised code does provide some gas savings

```
$ forge snapshot --diff
[...]
Overall gas change: -322060 (-0.105%)
```

we prefer not to change the current implementation, as it would introduce tight coupling between `CollateralLibrary` and `Deposit`. This may pose risks in the future if we ever change the implementation of `Deposit.sol` but forget to change that of `CollateralLibrary`, which could theoretically introduce bugs allowing the minting of collateral tokens.

Spearbit: Acknowledged but not reviewed.

5.6 Informational

5.6.1 Early self liquidation profitability is dependent on the time value of creditor's positions

Severity: Informational

Context: SelfLiquidate.sol#L31-L44

Description: Self liquidations are mostly a last resort to allow creditors to reclaim their credit and realize a loss immediately instead of waiting for the protocol to perform a standard liquidation at a loss to make them whole. However, there are some considerations to make in regards to the time value of money.

Debt position's must be sufficiently collateralized to avoid all forms of liquidations, although the risk assessment is made on minted debt which includes `debtPosition.faceValue`. To note, `debtPosition.faceValue` includes future interest that must be paid by the borrower to the loan's creditors.

Effectively, self-liquidations may become profitable for creditors because even though `debtInCollateralToken > assignedCollateral` is at a loss in nominal terms, the amount of collateral claimed exceeds the current value of their credit according to the specified yield curve.

Recommendation: Some considerations need to be made if this should be considered expected behaviour or if `assignedCollateral` in `executeSelfLiquidate()` should be adjusted down in real terms. Otherwise, further documentation outlining this odd behaviour should be noted. Similar behaviour also applies to standard liquidations in that creditors receive back their `principal + interest` earlier than expected.

Size: This is the intended behavior of the protocol.

Spearbit: Downgraded the severity to informational. Acknowledged but not reviewed.

5.6.2 Inconsistent naming of non-transferrable borrow token

Severity: Informational

Context: Initialize.sol#L240-L250

Description: Upon initializing a pool, *two* non-transferrable tokens are deployed. One for the borrow token and another for the collateral token. The naming of the collateral token concatenates the name and symbol of the underlying token to the non-transferrable token. However, this is not done for the borrow token.

Recommendation: Ensure there is consistency with the name/symbol of the non-transferrable tokens.

Size: Fixed in commit c1cd0ab6.

Spearbit: Acknowledged but not reviewed.

5.6.3 Price feed should take the `max` of the two oracles

Severity: Informational

Context: PriceFeed.sol#L50-L52

Description: Some considerations to make in regards to how the price feed currently functions. The `decimals` value is configured within the constructor of `PriceFeed.sol` and effectively defines the decimals of the price returned by `PriceFeed.getPrice()`. To make things simpler in design, `decimals` could be defined as a constant of `18` such that `getPrice()` consistently returns a price that minimizes rounding and is compatible with the current implementation of the Size Lending codebase.

Recommendation: Replace `decimals` with a constant of `18` to simplify the design of the price feed contract.

Size: Fixed in commit 131d3a93.

Spearbit: Acknowledged but not reviewed.

5.6.4 Variable rate hook calculations revert when the result is negative, while have to be floored at zero per protocol documentation

Severity: Informational

Context: YieldCurveLibrary.sol#L69-L74

Description: Resulting interest rates in user defined offers can be negative because of the variable rate hooks. Per docs the variable rate hook rate results should be floored at zero. Current implementation reverts when the result is negative. The reverting, i.e. failing when user defined curve can't be met, is a more desired behavior as adding any logic, such as flooring, to user preferences can be not fully expected (it might be the case that it's not even straightforward to fully accommodate such flooring, e.g. it's not a smooth as a function) and can alter user side strategies this way, possibly leading to user losses.

Recommendation: Consider updating the docs to match the current behavior.

Size: Fixed.

Spearbit: Acknowledged but not reviewed.

5.6.5 Total debt approach is not aligned with the classic definition of collateralization ratio

Severity: Informational

Context: LoanLibrary.sol#L75-L77

Description: `overdueLiquidatorReward` and `repayFee` should not be a part of collaterization, i.e. they might go with `1` multiplier, but not with `CR` multiplier. The reason is that both will be taken conditionally in the future, so cannot be treated as a part of the current borrower's debt. This way `CR` numbers protocol uses do not align with classic definition of `CR = Value(Collateral) / Value(Debt)`, which can cause discrepancies in users' risk assessment processes.

Recommendation: Consider either moving to origination fees, which are taken on position creation and so do constitute a part of the debt, or posting the collateral requirements for the future fees with the multiplier of `1`, only reserving the corresponding funds.

Size: Fixed

1. Commit 420b4061 replaces `repayFee` by `swapFee`, which is charged when a loan is generated.
2. Commit bfdabea4 removes `overdueLiquidatorReward` and updates the liquidation algorithm to reward liquidators based on the `faceValue`, not on the auto-assigned collateral, in order to mitigate another issue:

   Imho the main problem here is the inconsistency at the junction point between the

   - the incentive for profitable liquidation at CR=130% and
   - the incentive for overdue liquidation when deltaTOvedue=0 meaning the loan has just become overdue Since profitable liquidation reward > overdue liquidation reward then for loans with CR > 130% but very close to it, this creates an incentive for liquidators to wait instead of acting The summary is that instead of giving the liquidator a share of the collateral remainder he gets a share of the face value, of course paid in collateral and taken from the collateral backing the loan. This gives the liquidator a reward that is independent of the CR at which they liquidate and the implications are
   - if they liquidate with a high CR, then the protocol and the borrower get more than now and if they liquidate with a low CR the protocol and the borrower get less than now, to allow the liquidator to get back the same amount (of course with the exception of when the CR is close to 100% but that case is discussed later) meaning also the liquidator makes more than now for low CRs
   - this incentive allow us to reward liquidator more than now when we need them more meaning when the CR is low, while now they are rewarded the most when we need them the least meaning when the CR is high and close to 130%

- this is still not an incentive for them to wait since: there is no advantage for them in waiting as the reward is constant as long as it is possible, and decreases linearly when CR approaches 100%, they just risk to lose the liquidation opportunity taken by another liquidator, including us with the liquidate with replacement if possible
- this is fair for the protocol and lender perspective since at low CRs, when they earn less than now, they have good reasons to do that: the protocol wants to minimize the risk of bad debt and the borrower does not deserve getting back some collateral
- Finally, we can enforce consistency with the overdue since we can also set the overdue liquidation reward being the same percentage of the credit, meaning at that point there is no more incentive in waiting for an overdue loan with CR > 130% of a small amount to cross CR=130% and earn way more

Spearbit: Acknowledged but not reviewed.

5.6.6 Simplification of `PriceFeed` contract

Severity: Informational

Context: PriceFeed.sol

Description: The PriceFeed contract calculates prices by taking the ratio of two oracle feeds. However, concerns have been raised about the innecesary complexity of this method, given that in the current version its gonna be working only with `ETH/USD` and `USDC/USD`. Furthermore, the precision loss due to the fixed decimal handling in the current implementation could lead to inaccuracies when lower decimals are used as highlighted in the test case `PriceFeed.t.sol#L107-L111`.

Recommendation: Adjust and simplify the implementation to ensure the decimal precision is consistent and optimally set based on the oracle's decimals. Also, considering only specific asset pairs (ETH/USD, USDC/USD) are to be supported, the contract could be simplified by hard-coding checks for the required decimals and ensuring both feeds are compatible. Further, adopting a simplified calculation model as proposed below could reduce complexity and potential errors:


```
diff --git a/src/oracle/PriceFeed.sol b/src/oracle/PriceFeed.sol
index 02fec49..507ea9e 100644
--- a/src/oracle/PriceFeed.sol
+++ b/src/oracle/PriceFeed.sol
@@ -2,7 +2,6 @@
 pragma solidity 0.8.23;

 import {AggregatorV3Interface} from
↪  "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";
-import {SafeCast} from "@openzeppelin/contracts/utils/math/SafeCast.sol";
 import {Math} from "@src/libraries/Math.sol";

 import {IPriceFeed} from "./IPriceFeed.sol";
@@ -19,11 +18,11 @@ import {Errors} from "@src/libraries/Errors.sol";
 ///      answer: ETH/USDC in 1e18
 contract PriceFeed is IPriceFeed {
     /* solhint-disable immutable-vars-naming */
+    /// @notice base is ETH/USD feed
     AggregatorV3Interface public immutable base;
+    /// @notice quote is USDC/USD feed
     AggregatorV3Interface public immutable quote;
     uint8 public immutable decimals;
-    uint8 public immutable baseDecimals;
-    uint8 public immutable quoteDecimals;
     uint256 public immutable baseStalePriceInterval;
     uint256 public immutable quoteStalePriceInterval;
     /* solhint-enable immutable-vars-naming */
@@ -53,8 +52,8 @@ contract PriceFeed is IPriceFeed {
```

```
         baseStalePriceInterval = _baseStalePriceInterval;
         quoteStalePriceInterval = _quoteStalePriceInterval;

-        baseDecimals = base.decimals();
-        quoteDecimals = quote.decimals();
+        require(base.decimals() == 8, "PriceFeed: invalid base decimals");
+        require(quote.decimals() == 8, "PriceFeed: invalid quote decimals");
     }

     function getPrice() external view returns (uint256) {
@@ -72,18 +71,6 @@ contract PriceFeed is IPriceFeed {
             revert Errors.STALE_PRICE(address(aggregator), updatedAt);
         }

-        price = _scalePrice(price, aggregator.decimals(), decimals);
-
-        return SafeCast.toUint256(price);
-    }
-
-    function _scalePrice(int256 _price, uint8 _priceDecimals, uint8 _decimals) internal pure returns
↪   (int256) {
-        if (_priceDecimals < _decimals) {
-            return _price * SafeCast.toInt256(10 ** uint256(_decimals - _priceDecimals));
-        } else if (_priceDecimals > _decimals) {
-            return _price / SafeCast.toInt256(10 ** uint256(_priceDecimals - _decimals));
-        } else {
-            return _price;
-        }
+        return uint256(price);
     }
 }
```

Size: Fixed in commit 131d3a93.

Spearbit: Acknowledged but not reviewed.

5.6.7 Missing `multicall` function, events and errors on `ISize` Interface

Severity: Informational

Context: Size.sol#L57-L60, ISize.sol#L31

Description: The `ISize` interface currently does not include definitions for events, errors, and the `multicall` method. These components are essential for comprehensive interface functionality and may impact developers who rely on `ISize` for building applications on top of Size.

Recommendation: Review and update the `ISize` interface to include the missing events, errors, and the `multicall` method.

Size: Fixed in commit 8da42354.

Spearbit: Acknowledged but not reviewed.

5.6.8 Repayments may be prevented if the variable pool `supply` reverts (e.g. due to caps)

Severity: Informational

Context: VariablePoolLibrary.sol#L31

Description (*issue raised by Size team*)**:** One design decision of Size is to apply all unlent money into a variable-rate loan pool for passive yield.

The issue is that when the pool prevents deposits, it can cause a Denial of Service on the protocol. If Aave's `supply` function reverts for any reason, such as a supply limit on their side, borrowers might be unable to repay their loans, and liquidators to liquidate underwater positions.

An underwater borrower can exploit this to avoid liquidations by, for example, depositing too many borrow tokens on Aave.

Looking into the data for Base, it seems like aUSDC currently has $8.1M USDC in deposits, with a cap of $60M. This means this attack would be rather capital-intensive, so not very likely. Still, by not fixing this, the protocol would have a big dependency on the well-functioning of Aave, which can be damaging especially if Size is successful.

Note: the discussion from this issue centered on the cap of the collateral token, and concluded that it should be removed altogether to avoid stopping borrowers from preventing liquidations. However, it did not mention the implicit cap from Aave, and how it could stop Size from working.

Recommendation: One idea is to store not only `scaledBorrowATokenBalance` by users, but also `underlyingBorrowTokenBalance`, which represents underlying tokens not deposited on a third-party protocol, which could still be used for repayments, etc...

Size: Acknowledged. We have decided to acknowledge this risk, as implementing a solution to handle this edge case can be quite error-prone and may still open doors to other attacks. We will provide a disclaimer to users explaining that our protocol relies on the well-functioning of Aave.

Spearbit: Acknowledged but not reviewed.

5.6.9 Documentation recommendations

Severity: Informational

Context: Documentation

Description: There are some points in the protocol documentation that look enhanceable.

Recommendation: In order of appearance:

- Lending: Instead `a replacement lender is appointed and the exiting lender pays` it can be `a replacement lender is appointed and this lender pays`.
- Illustration of borrowing against receivables: Shouldn't it be `Feb 2, 2024` at Step 4? There looks to be no reason for introducing a delay and adding `1 day` to Lender B's effective term.
- Illustration of overdue borrower moved to variable pool: For Step 2: in (1) should be `Borrower starts paying variable rate on 535 USDC` instead of `Lender starts paying variable rate on 535 USDC`.
- Illustration of borrower liquidation w replacement borrower: For Step 3: should be `and a bid from the Borrowing Bid Order Book is filled` instead of `and a bid from the Borrowing Bid Offer Book is filled`.
- Governance variables: In raw amount terms `overdueLiquidatorReward` is `10e6`, while `minimumCreditBorrowATokenis` `5e6`.
- Credit position: `The are created either during the creation of a loan` should be `They are created ...`
- Borrowing market order: It looks like `FOL` isn't defined anywhere, shouldn't `FOL` be something like `debt and credit positions`?

- Automatic collateral assignment: `the totoal outstanding debt of the loan` should be `the total outstanding ...`
- Liquidation mechanisms:
  1) In `3.5.3.2 Self Liquidation`, Proof: `(x * fees) / V` term looks to be omitted in `CR after is` expression.
  2) Also there, in `So we have that` the first term should be `C/(V+)` instead of `C/V`
  3) In `3.5.3.3 Liquidation with replacement` the issuance value should be `V/(1+r)` instead of `V/r`

Size:

It looks like FOL isn't defined anywhere, shouldn't FOL be something like debt and credit positions?

In a previous version of our design, a loan was stored in a data structure called FOL (first-order loan), while a loan exited to another user was stored in a data structure called SOL (second-order loan). We decided to refactor it and split into `DebtPositions` and `CreditPositions`, to make it clearer. This means this part of the docs is outdated.

Spearbit: Acknowledged but not reviewed.

6 Appendix

Below are some issues that have been independently identified by te Size team while running invariant tests and conducting an internal review.

6.1 Repayments are prevented during collateral price drops due to an incorrect CR check

Context: Size.sol#L199

Description: In `Size.repay`, the `state.validateUserIsNotUnderwater(msg.sender)` check was introduced to force borrowers to repay their debt in full, assuming they have more than one loan. However, due to the composable nature of the protocol functions, this introduces a bug where underwater borrowers may not repay at all, since they can still have CR < 130% after a repayment, even if they have an incentive to repay.

This bug was found with a DoS check on invariant tests asserting that "repay never fails except for input validation conditions."

Recommendation: Remove the CR check.

6.2 Repayments may be prevented if the variable pool `supply` reverts (e.g., due to caps)

Context: VariablePoolLibrary.sol#L31

Description: One design decision of Size is to apply all unlent money into a variable-rate loan pool for passive yield.

The issue is that when the pool prevents deposits, it can cause a Denial of Service on the protocol. If Aave's `supply` function reverts for any reason, such as a supply limit on their side, borrowers might be unable to repay their loans, and liquidators to liquidate underwater positions.

An underwater borrower can exploit this to avoid liquidations by, for example, depositing too many borrow tokens on Aave.

Looking into the data for Base, it seems like aUSDC currently has $8.1M USDC in deposits, with a cap of $60M. This means this attack would be rather capital-intensive, so not very likely. Still, by not fixing this, the protocol would have a big dependency on the well-functioning of Aave, which can be damaging especially if Size is successful.

Note: the discussion from this issue centered on the cap of the collateral token, and concluded that it should be removed altogether to avoid stopping borrowers from preventing liquidations. However, it did not mention the implicit cap from Aave, and how it could stop Size from working.

Recommendation: One idea is to store not only `scaledBorrowATokenBalance` by users, but also `underlyingBorrowTokenBalance`, which represents underlying tokens not deposited on a third-party protocol, which could still be used for repayments, etc.

6.3 Possible DoS due to WadRay rounding

Context: VariablePoolLibrary.sol#L66-L68

Description: In `VariablePoolLibrary`, the WadRayMath library from Aave is used to convert from scaled balances to unscaled balances by applying `rayMul` and `rayDiv` operations to users' deposits. The issue is that these functions round to the nearest wad, and as a result, can be inconsistent in the rounding direction, leading to a denial of service.

This bug was found with a DoS check on invariant tests asserting that "claim never fails except for input validation conditions". The steps the fuzzer found were not intuitive, but the underlying issue is evident. Here's the sequence it found:

1. Alice borrows from herself with a face value of 1521433770. We have 1 debt position and 1 credit position with FV=Credit.

2. Alice compensates 5000001 from herself. Now we have 1 debt position with updated FV=1516433769 and 2 credit positions, one with Credit=1511433768 and another with Credit=5000001. Still, SUM(FV) = SUM(Credit), so nothing wrong here.
3. The liquidity index is updated from 1 RAY to 1020354318670279567035913333.
4. Alice repays her loan. Now, the Size contract holds her 1516433769 in cash, her debt position is updated to FV=0, and the 2 credit positions from step 2 are claimable.
5. Alice claims the 1st credit position. The size contract transfers 1511433768 to her, but then something happens. The problem is that when trying to claim the 2nd credit position, borrowATokenBalanceOf(address(this)) returns 5000000. That means, 1 wei less than what she is claiming, preventing the Size contract from giving the lender the money from the repaid position. Her 2nd credit position is thus unclaimable.

The example the fuzzer found is not realistic because she'd be paying fees for borrowing from herself, but the same could happen in different circumstances.

Recommendation: Instead of using `WadRayMath`, always round down the scaled balances, such as with `Math.mulDivDown`. After applying this change, the fuzzer could not find the same DoS after 48 hours running. Note: by not using `WadRayMath` for scaled operations, users potentially lose 1 due always to rounding down.

Another idea might be allowing partial claims, which may prevent future unforeseen rounding issues. The challenge is that we'd have to support and think about the consequences of partial claims. Please advise on this.

6.4 `Size.liquidateWithReplacement` incorrectly calculates protocol fees after replacement

Context: LiquidateWithReplacement.sol#L122

Description: In `LiquidateWithReplacement.executeLiquidateWithReplacement`, the new borrower's total debt tracker is set to the previous loan's total debt. This is incorrect, since after the replacement, the issuance value changes, and so the repay fee changes. As a result, two important invariants of the system break:

- The sum of all debt positions' debt is equal to the total supply of the debt token.
- The sum of all debt positions' debt of a user is equal to the debt token's balance of that user.

Recommendation: Use `debtPosition.getTotalDebt()` instead.

6.5 Charging fees of underwater borrowers on `repay` disincentivizes repayment

Context: Repay.sol#L50

Description: In `Repay.executeRepay`, the repay fees are always charged, regardless of the borrower being healthy, underwater (CR < 130%), or heavily underwater (CR < 100%), but in self-liquidation (so CR < 100%), no fees are charged. Should they be charged in case of repayment?

Imagine a situation where the total debt of a loan is FV=100 + Fees=5 and the auto-assigned collateral value is 102.

The CR=97% so the borrower is heavily underwater meaning he has an undercollateralized loan. At this point, he has no incentive to repay.

However, if the protocol does not charge fees in this case, then the borrower can repay 100 and get back 102 of collateral, so he would have an incentive to repay, and we avoid lenders having to go for self-liquidations.

Also, from the point of view of consistency, imagine, for the sake of simplicity, there is only one lender: if he self-liquidates himself (which is possible since CR < 100%), then he takes 100% of the auto-assigned collateral meaning 102 and he does not take any loss; on the contrary, he makes a profit.

Recommendation: Do not charge fees during `repay` if the borrower is underwater. Please help us review this recommendation.

6.6 Repay is prevented by accounts other than the borrower

Context: Repay.sol#L34-L36

Description: In `Repay.validateRepay`, if the sender is not the borrower, the operation reverts. It seems like this check is not necessary and could prevent the overall health of the system from improving.

Recommendation: Allow any address to repay a loan. Please let us know if this could cause any issues.

6.7 Borrow token caps may prevent repayment and liquidations

Context: Size.sol#L131

Description: Since caps are enforced at each deposit operation on Size, a liquidator may run into a revert when trying to deposit USDC to liquidate, or a depositor when trying to repay.

Underwater borrowers may leverage this to avoid getting liquidated.

Recommendation: One solution is to ignore caps during debt reduction event. The challenge is that our protocol enforces the separation between deposits and other actions, which would require a significant refactor to accept incoming tokens in repay/liquidate, for example.

An alternative is to validate caps only at the end of a batch (multicall) operation:

- The cap on borrow tokens is only applied to underlying deposits, i.e., it ignores yield from Aave.
- At the beginning of the multicall, we set a lock/flag to ignore all caps. At the end of the multicall (ie, as a result of a deposit+repay or deposit+liquidate), we only check the caps if the repaid amount (ie, by analysing the decrease in debt token total supply) is at least equal to the underlying inflow (ie, by analysing the change in underlying token total deposits).

Please let us know what you think of this proposal, especially taking into account the thread opened here.

6.8 Missing input validation in some functions could cause unexpected reverts

Description:

- `borrowerExit` was not checking that the borrower has an open borrow offer nor that the loan was active.
- `compensate` was not checking that the two credit position IDs were the same, nor that the amount to compensate was lower than the available credit.
- `lendAsMarketOrder` was not checking the minimum credit amount. This issue had been raised and acknowledged on a previous audit, but not having this input validation would lead to the undesired effect of unexpected reverts, so we decided to amend it.
- `borrowAsMarketOrder` was not checking the minimum credit amount. This issue had been raised and acknowledged on a previous audit, but not having this input validation would lead to the undesired effect of unexpected reverts, so we decided to amend it.