SOLIDIFIED

Audit Report for Size v1 - March 26, 2024

Summary

Audit Report prepared by Solidified covering Size v1.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on February 28, 2024, and the results are presented here.

Audited Files

The source code has been supplied in a private GitHub repository:

https://github.com/SizeLending/size-v2-solidity

Commit hash: `1187aed0699d4bb373f00413a25bbf32771247df`

Fixes received at commit: `c1362d5488e34365e975fa1980d7d0d8a0502c04`

File list:

```
src
├── Size.sol
├── SizeStorage.sol
├── SizeView.sol
├── interfaces
│   └── ISize.sol
├── libraries
│   ├── ConversionLibrary.sol
│   ├── Errors.sol
│   ├── Events.sol
│   ├── Math.sol
│   ├── fixed
│   │   ├── AccountingLibrary.sol
│   │   ├── CapsLibrary.sol
│   │   ├── CollateralLibrary.sol
│   │   ├── LoanLibrary.sol
│   │   ├── OfferLibrary.sol
│   │   ├── RiskLibrary.sol
│   │   └── UserLibrary.sol
```

```
│   │   ├── YieldCurveLibrary.sol
│   │   └── actions
│   │       ├── BorrowAsLimitOrder.sol
│   │       ├── BorrowAsMarketOrder.sol
│   │       ├── BorrowerExit.sol
│   │       ├── Claim.sol
│   │       ├── Compensate.sol
│   │       ├── Deposit.sol
│   │       ├── LendAsLimitOrder.sol
│   │       ├── LendAsMarketOrder.sol
│   │       ├── LiquidateLoan.sol
│   │       ├── LiquidateLoanWithReplacement.sol
│   │       ├── Repay.sol
│   │       ├── SelfLiquidateLoan.sol
│   │       └── Withdraw.sol
│   ├── general
│   │   └── actions
│   │       ├── Initialize.sol
│   │       └── UpdateConfig.sol
│   └── variable
│       └── VariableLibrary.sol
├── oracle
│   ├── IMarketBorrowRateFeed.sol
│   ├── IPriceFeed.sol
│   ├── MarketBorrowRateFeed.sol
│   ├── PriceFeed.sol
│   └── VariablePoolPriceFeed.sol
├── proxy
│   └── Vault.sol
└── token
    └── NonTransferrableToken.sol
```

Intended Behavior

The contract implements a fixed-rate lending protocol with unified liquidity.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	High	-

SOLIDIFIED

Audit Report for Size v1 - March 26, 2024

Issues Found

Issue #	Description	Severity	Status
1	Compensating a loan results in the lender being unable to claim and the borrower unable to repay and exit the loan	Critical	Resolved
2	Inability to liquidate compensated and receivable loans	Critical	Resolved
3	Second-order loan (SOL) self-liquidation is malfunctioning due to using the wrong borrower to determine the collateral ratio	Critical	Resolved
4	Early exiting a loan overcharges the repayment fee, and subsequently repaying a loan results in the debt not being fully repaid and the lender being unable to claim the funds	Critical	Resolved
5	No time-weighting and timeout for the Aave borrowing rate potentially lead to rate manipulation, which might result in unintentional execution of orders	Critical	Resolved
6	Negative correlation between the reference rate and the lending rate on Size leads to collapsing yield curves during systemic events, reverting transactions and/or users being forced into trades	Critical	Resolved
7	Lending and borrowing functionality is susceptible to front-running	Major	Resolved
8	Self-liquidating loans round up the burned repay fee, resulting in potentially insufficient debt tokens for subsequent SOL self-liquidations	Major	Resolved
9	Inability to pause the contract in case of an emergency	Major	Resolved

10	Borrower loses entire collateral if an overdue loan is liquidated	Major	Resolved
11	The risk fund offers insurance against high collateral volatility when one lends to oneself	Major	Acknowledged
12	Order book spoofing allows grief attacks and can cause systemic risk and sudden drops of liquidity	Minor	Acknowledged
13	Suboptimal liquidation incentives put a higher burden on the insurance fund and lead to systemic risk	Minor	Acknowledged
14	Repayment fees charged during compensation potentially put borrowers at risk of liquidation	Minor	Resolved
15	Overdue loans moved to Aave's variable pool can be at risk of liquidation	Minor	Resolved
16	Loans with a maturity date set to the current block timestamp can be immediately liquidated	Minor	Resolved
17	Overdue liquidation fee potentially exceeds the borrower's collateral balance, preventing the timely liquidation of the loan	Minor	Acknowledged
18	Size protocol admin can potentially liquidate all positions by setting crLiquidation higher than all borrower collateral ratios	Minor	Acknowledged
19	Size protocol admin operator can potentially liquidate all positions by assigning a malicious oracle	Minor	Acknowledged
20	Withdraw event can emit an incorrect withdrawal amount	Minor	Resolved
21	aaveOracle.getAssetPrice calls ChainLink AccessControlledOffchainAggregator.latestAnswer, which is deprecated and not recommended to be used	Minor	Resolved
22	Lack of a secondary fallback price feed oracle	Minor	Acknowledged

23	Validate that the Oracle address is an IPriceFeed contract	Minor	Resolved
24	Claiming a loan that is already claimed returns a misleading LOAN_NOT_REPAID error	Minor	Resolved
25	When a vault aToken balance is insufficient, the returned error is PROXY_CALL_FAILED, which is unclear to the user	Minor	Resolved
26	It is not possible to cancel lend limit orders	Minor	Resolved
27	In the validateCompensate function, the repaid state of the loan to compensate is checked twice	Note	Resolved
28	Add a fallback price discovery mechanism against Oracle manipulation	Note	Acknowledged
29	Validate that configured addresses are distinct during initialization	Note	Acknowledged
30	Validate that a borrow order is below the dust limit in the validate function	Note	Acknowledged
31	Liquidations revert if they are not profitable for the liquidator	Note	Acknowledged
32	USDC stablecoin centralization risk	Note	Acknowledged
33	Avoid using the latest Solidity compiler version to reduce compiler bug risks	Note	Resolved
34	In Size.sol, the State struct is imported twice	Note	Resolved
35	In Errors.sol, the @title attribute is incorrectly marked as “Events”	Note	Resolved
36	All possible event parameters should be indexed	Note	Resolved
37	AccountingLibrary.chargeRepayFee could be renamed or split into two functions	Note	Resolved
38	RiskLibrary.getMinimumCollateralOpening function is never used in the protocol	Note	Resolved

39	UserLibrary definition is empty and can be removed	Note	Resolved
40	The White Paper states that rates can be negative while the protocol defines YieldCurve.rates values as unsigned integers	Note	Resolved
41	Empty test definitions	Note	Acknowledged
42	Typo in VariableLibrary.depositUnderlyingBorrowTokenToVariablePool in @dev docs	Note	Resolved
43	In the White Paper, a borrower can specify N>=1 LLOs while the protocol only allows a single LLO to be specified	Note	Acknowledged
44	Empty function definition for validateUpdateConfig	Note	Acknowledged
45	Miscellaneous Comments	Note	

Critical Issues

1. Compensating a loan results in the lender being unable to claim and the borrower unable to repay and exit the loan

In `Compensate.sol`, the `executeCompensate` function sets the liquidity index, `liquidityIndexAtRepayment`, to the current liquidity index retrieved from the forked Aave v3 variable pool if the borrower's available credit fully compensates the repaid loan's debt. This is to ensure that the repaid loan continues to accrue interest on the escrowed ATokens until the funds are claimed by the lender later.

However, instead of setting the liquidity index of the repaid loan `loanToRepay`, it is set for the compensated loan, `loanToCompensate`. As a result, the lender cannot claim the repaid loan's funds, as the calculated `claimAmount` in the `executeClaim` function in `Claim.sol` will be zero, rendering the funds locked in the contract.

Moreover, repaying the borrowed tokens from the compensated loan, whose credit was fully utilized for compensating the debt and thus having the status `LoanStatus.CLAIMED` is also malfunctioning due to the `validateRepay` function in `Repay.sol` preventing a loan with the status `LoanStatus.CLAIMED` from being repaid. Please note that this issue also applies to using receivable loans as credit when borrowing via the `borrowAsMarketOrder` function.

To summarize, the aforementioned issues result in the lender of the repaid loan being unable to claim funds and the borrower of the compensated loan being unable to repay the borrowed tokens, exit the loan, and receive the collateral.

We emphasize that fixing this issue by setting the liquidity index for the repaid loan, `loanToRepay`, would allow the lender of the repaid loan to prematurely claim the repaid funds before the compensated loan is repaid and tokens are actually available. This would result in receiving funds from other lenders, preventing them from claiming.

Recommendation

We recommend correctly setting the liquidity index for the repaid loan and updating the `validateRepay` function to allow repaying a loan with the status `LoanStatus.CLAIMED` if the loan's debt is fully compensated.

Status

Resolved

2. Inability to liquidate compensated and receivable loans

The `validateLiquidateLoan` function in `LiquidateLoan.sol` ensures that the loan is liquidatable by calling the `isLoanLiquidatable` function in `RiskLibrary.sol`.

However, this function only considers first-order loans (FOL) with the status `LoanStatus.ACTIVE` or `LoanStatus.OVERDUE` as liquidatable, preventing a compensated or receivable loan from being regularly liquidated and requiring the lender to self-liquidate instead. But, due to another issue explained in the separately reported finding, "Second order loan (SOL) self-liquidation is malfunctioning due to using the wrong borrower to determine the collateral ratio", self-liquidation by the lender is also malfunctioning.

As a result, the lender of the compensated or receivable loan does not receive the pro-rata collateral amount, manifesting as a loss of funds and bad protocol debt.

A test case reproducing the issue is provided in Appendix 1.

Recommendation

We recommend allowing the liquidation of loans with the `LoanStatus.CLAIMED` status by amending the `isLoanLiquidatable` function in `RiskLibrary.sol` to include `LoanStatus.CLAIMED` in the status check.

Status

Resolved

3. Second-order loan (SOL) self-liquidation is malfunctioning due to using the wrong borrower to determine the collateral ratio

---

The `isLoanSelfLiquidatable` function in `RiskLibrary.sol` determines if the lender can self-liquidate the specified loan, either a first-order (FOL) or secondary-order loan (SOL). This is achieved by checking if the loan's borrower, `loan.generic.borrower` is liquidatable and if the loan's status is either `LoanStatus.ACTIVE` or `LoanStatus.OVERDUE`.

However, if the provided loan is a SOL, e.g., due to compensation of a loan or using a loan as a receivable, using `loan.generic.borrower` to determine the borrower collateral ratio is incorrect as this borrower represents the borrower of the SOL, not the FOL.

For example, consider the following scenario:

- Loan 1, with Bob as the borrower and Alice as the lender
- Loan 2, with Alice as the borrower and Charles as the lender

Alice uses the credit (i.e., the future cash flow) of loan 1 to compensate and repay loan 2. This creates a second-order loan, loan 3, with Alice as the borrower and Charles as the lender. Due to market volatility, Bob's collateral ratio falls below the liquidation threshold. Loan 1 can not be liquidated due to its status being `LoanStatus.CLAIMED`, an issue separately reported in the finding "Inability to liquidate compensated and receivable loans". This leaves Charles, the lender of loan 2, with only the option to self-liquidate loan 3 to cut his losses.

As explained above, the `isLoanSelfLiquidatable` function uses the borrower of the provided loan 3, Alice, to determine if the borrower is liquidatable. However, this is incorrect as the correct borrower would actually be Bob, the borrower of the SOL's corresponding FOL, loan 1. Consequently, the health of the loan is incorrectly determined, preventing the lender from self-liquidating the loan to receive the pro-rata collateral amount.

A test case reproducing the issue is provided in Appendix 2.

Recommendation

We recommend retrieving the FOL of the given loan in the `isLoanSelfLiquidatable` function and using the borrower of the FOL to determine if the user is liquidatable.

Status

Resolved

4. Early exiting a loan overcharges the repayment fee, and subsequently repaying a loan results in the debt not being fully repaid and the lender being unable to claim the funds

---

The early exit borrower fee is applied to the initial borrower when exiting a loan to a new replacement borrower before maturity. In addition, the initial borrower pays the discounted cash flow value based on the interest rate of the replacement borrower by transferring the borrowed ATokens to the new borrower. The number of tokens transferred, `amountIn`, calculated in the `executeBorrowerExit` function in `BorrowerExit.sol`, is based on the loan's debt, including the face value and the repayment fee.

However, the initial borrower's outstanding repayment fee is not prorated based on the elapsed time. Instead, it is based on the full loan maturity while the replacement borrower has the loan duration adjusted and the repayment fee reduced. In addition, the interest rate should not be applied on top of the repayment fee as this leads to overcharging the initial borrower.

Consequently, the loan's full debt can not be repaid at maturity due to incorrectly calculating the fee, locking the borrower's collateral, and preventing the lender from claiming the funds.

For example, consider a loan with an issuance value of 100, an annual interest rate of 10%, a maturity of 12 months, and a repayment fee of 10% (10). The face value of the loan is 110, and the repayment fee amounts to 10, resulting in a total debt of `110 + 10 = 120`. After half of the loan duration, 6 months, the borrower early exits to a replacement borrower with the same

interest rate of 10%. The discounted cash flow is calculated as `120 * 100/(100 + 10/2) = ~114.29` and transferred to the new borrower.

At loan maturity, the replacement borrower would have to repay the entire debt (including the repayment fee) of 120 to exit the loan. However, as the loan's start date was adjusted during the early exit, effectively halving the repayment fee to 5, the borrower only has to repay `110 + 5 = 115` instead of 120. This leaves a debt of 5 remaining, without being able to repay, resulting in the inability to exit the loan and receive the collateral fully.

Ultimately, the initial borrower is overcharged, the new borrower is undercharged and unable to exit the loan, and the lender can not claim the funds.

An attacker can use these shortcomings by exiting to themselves multiple times to reduce their debt burden significantly. As the overcharged amount is insufficient to compensate for the undercharged amount, the attacker steals funds from the lender.

Recommendation
We recommend revisiting the `amountIn` calculation in `executeBorrowerExit` to ensure the initial and the replacement borrowers are charged correctly.

Status
Resolved

5. No time-weighting and timeout for the Aave borrowing rate potentially lead to rate manipulation, which might result in unintentional execution of orders.

The Size protocol takes the current Aave borrowing rate as given - i.e., the actual pool rate on Aave is used as an input to calculate short and long-term interest rates on Size. However, the Aave rate can vary substantially depending on the pool utilization and can be manipulated by large capital injections and withdrawals.

We specifically mention large capital injections here in addition to flash loans, since attackers might find it worthwhile manipulating the price for longer periods - e.g., hours - on Aave. If the total interest paid on Aave is lower than the saved interest rate on Size. If attackers try to manipulate prices downwards by sending liquidity to Aave, they receive interest for doing so. Hence, compared to other price manipulation attacks, that have to be backed by actual sales, such an attack might be less costly.

Multiple arbitrage-like trades of the following form could also leverage an exploit:

1. Manipulate the Aave rate slightly
2. Borrow at the lowest point of the yield curve on Size
3. Deploy the liquidity to Aave to lower the higher sections of the yield curve further
4. Borrow more offers on Size at a reduced yield.

As a result, orders might be executed against the intention of the borrower or lender.

Together with the issue of the negative correlation of rates, another issue arises: A malicious user could borrow from Aave such that the rate increases, which again might enter the rate calculation formula with a negative sign, to then borrow at lower rates from Size. Or deploy capital to Aave in order to increase the lending rates on Size and the protocol it illiquid. As we assume that this will be fixed, we do not raise a separate issue.

Furthermore, we highlight that the governance of the competitor Aave might have an incentive to manipulate or sabotage Size.

Recommendation
We recommend implementing various measures against such attacks:

1. Implementing a time-weighted average price and stale rates to prevent flash loans or multi-block manipulations of the lending rate
2. The waiting period should be significantly longer than the price oracles of assets. Note: This decision influences the minimal lending period, i.e., that lending within the TWAP time shall not be possible with reference to a market rate

3. Implementing a logic that deals with Aave rate time-outs
4. Fallback rate

Status
Resolved. The client states that they have decided to implement an off-chain TWAP Oracle to avoid manipulation attacks.

6. Negative correlation between the reference rate and the lending rate on Size leads to collapsing yield curves during systemic events, reverting transactions and/or users being forced into trades

---

It is currently possible to submit negative and positive rate multipliers and positive constants to parameterize the yield curve.

- $-1\,Aave + 3\%$ and $3\,Aave + 1\%$ are allowed rates
- $Aave - 2\%$ is not allowed
  where $Aave$ denotes the rate on Aave

In addition to what is described in the documentation, the Size team explained that the constant should be allowed to take on negative values, and the negative multipliers will remain. This is because rates that are parameterized to be a fraction of the Aave rate lower than the Aave rate will be implemented via the UI as negative multipliers - e.g., 10% below the Aave rate will be implemented by $-0.1\,Aave$.

There is little economic justification to have inversely parametrized interest rates. In the worst case, negative multipliers can lead to the collapse of rates on Size if there is a systemic event or Aave governance decision that leads to unconventionally high interest rates.
As a result, transactions might revert, or users might be forced into trades at very low rates while the rates on Aave are high. This is particularly problematic if not only educated users set the negative multiplier, but it is set through the UI to implement fractional deviations.

A 10% cheaper rate than Aave can be implemented by setting the rate to $0.9\ Aave$.
The issue is exacerbated as it is not possible to cancel lending orders completely other than removing the liquidity from the protocol.

Recommendation
We recommend allowing only positive multipliers.

Status
Resolved

Major Issues

7. Lending and borrowing functionality is susceptible to front-running

Offers for lending and borrowing are submitted on either side of the order book as offers to lend or bids to borrow. For example, a lender can lend funds to a specific borrower with a corresponding bid in the order book via the `lendAsMarketOrder` function in `Size.sol`.

However, the borrower can change the terms of the borrow offer at any time, potentially front-running the lender and changing the terms to the lender's disadvantage. For instance, the borrower can significantly lower the borrowing rate, reducing the lender's yield.

Conversely, the lender can change the loan offer at any time and update the yield curve to receive a higher yield, acting to the borrower's disadvantage.

Recommendation

As a short-term solution, we recommend adding some form of front-running protection to the `lendAsMarketOrder`, `borrowAsMarketOrder`, and `borrowerExit` functions, such as a parameter that allows the caller to specify the maximum or minimum acceptable borrow rate.

Long term, we recommend revising the logic such that it locks up the liquidity behind an order for a valid duration, as grief attacks are still possible and might lead to systemic risk.

Status
Resolved

8. Self-liquidating loans round up the burned repay fee, resulting in potentially insufficient debt tokens for subsequent SOL self-liquidations

---

In `SelfLiquidateLoan.sol`, the `executeSelfLiquidateLoan` function calls `chargeRepayFee` to charge the repayment fee and to decrease the loan's issuance value by the repayment amount. The repay fee, `repayFee`, is calculated by the `partialRepayFee` function and rounded up due to using `Math.mulDivUp`, resulting in always charging, that is burning, at least `1 wei` of the debt token as the repay fee.

However, if both a second-order (SOL) and the corresponding first-order loan (FOL) are self-liquidated by the respective lenders, the repay fee of at least `1 wei` is burned for both loans, resulting in insufficient debt tokens to burn for the second self-liquidation attempt.

Please note that self-liquidating a SOL is currently malfunctioning, as reported in "Second-order loan (SOL) self-liquidation is malfunctioning due to using the wrong borrower to determine the collateral ratio", and thus this issue only becomes relevant if the self-liquidation mechanism is fixed as recommended. Nevertheless, we classify this issue as Major to prevent potential issues caused by an oversight while mitigating the self-liquidation issue.

A test case reproducing the issue is provided in Appendix 3.

Recommendation

We recommend revising the `chargeRepayFee` function to only burn the repay fee debt tokens if the borrower's remaining debt token balance is greater than the repay fee.

Status

Resolved

9. Inability to pause the contract in case of an emergency

In `Size.sol`, the `PAUSER_ROLE` is granted to the contract owner during the contract instantiation via the `initialize` function, presumably to allow the owner to pause and unpause the contract in case of an emergency.

However, the contract does not implement the required functions to pause and unpause the contract by calling the internal `_pause` and `_unpause` functions of the inherited `PausableUpgradeable` contract. Consequently, pausing and unpausing critical functionality is not possible, resulting in the inability to halt the contract in case of an emergency.

Recommendation

We recommend implementing a `pause` and `unpause` function in the `Size` contract, which calls the internal `_pause` and `_unpause` functions of the inherited `PausableUpgradeable` contract, respectively.

Status

Resolved

10. Borrower loses entire collateral if an overdue loan is liquidated

In `LoanLibrary.sol`, the `getFOLAssignedCollateral` function is used to calculate the amount of collateral assigned to a loan. However, if a borrower has a healthy collateral ratio (CR) and has a loan that is in an `LoanStatus.OVERDUE` state, they will lose their entire collateral

upon liquidation. This is because the `collateralAmount` used is based on the `getFOLAssignedCollateral,` which can be the borrower's entire collateral.

Scenario:

1. Borrower deposits 100 WETH as collateral
2. Borrower takes a loan of 10 USDC (very healthy CR).
3. Borrower does not repay, and the loan moves to `LoanStatus.OVERDUE` state (with a very healthy CR)
4. The loan is liquidated and follows the loan overdue path, resulting in a call to `VariableLibrary.moveLoanToVariablePool`
5. The amount of collateral to transfer from the borrower to the variable pool is calculated using `getFOLAssignedCollateral`, which returns the borrower's entire collateral amount held in the protocol by the borrower at that time.
6. The borrower's entire collateral is transferred to the variable pool via the vault proxy.
7. The borrower cannot claim their surplus collateral as it is now locked in the vault proxy.

A test case reproducing the issue is provided in Appendix 4.

Recommendation

We recommend adjusting the collateral required to cover the debt for `LoanStatus.OVERDUE` loans using a similar approach as used in the `_executeLiquidateLoanTakeCollateral` function in `LiquidateLoan.sol` to calculate the `debtInCollateralToken` .This converts the USDC debt to the equivalent value of the debt in collateral tokens using the price oracle.

Status

Resolved

11. The risk fund offers insurance against high collateral volatility when one lends to oneself

This fund will execute unprofitable liquidations based on the internal business logic of the Size protocol, executed by bots. In the process, the lender is reimbursed with their lending amount, all collateral is seized, and the borrower's debt is annulled.

However, insiders or attackers who manage to decipher the business logic by observing insurance fund liquidations can exploit this knowledge to create a hedge against the high volatility of the collateral asset. Given this mechanism carries an implicit insurance value, it can be regarded as a free hedge at the expense of other users of the protocol.
The hedge operates as follows: Should one anticipate or encounter significant volatility of the collateral asset, you lend money to yourself. If the market trends favorably and the asset's value does not drop below the liquidation threshold, you incur only the transaction fees. If the value decreases below 100%, your losses are limited to the capped liquidation expenses.

We classify this issue as only major because while implicitly, the fee-earnings of the protocol are used for unintended and highly specific purposes, pursuing such a strategy requires strong financial engineering and the right market conditions.

Recommendation
We recommend compensating lenders not beyond the actual value of the collateral. Furthermore, we recommend moving away from reliance on internal business logic for liquidations in favor of a transparent, rule-based policy to avoid any form of insider trading.

Status
Acknowledged. The client states that they understand the limitations of the insurance fund and we will re-evaluate how to make it more sustainable in a future update.

Minor Issues

12. Order book spoofing allows grief attacks and can cause systemic risk and sudden drops of liquidity

For any incoming `borrowAsMarketOrder` a frontrunning lender can always make the order fail by withdrawing all liquidity. This issue persists even if the separately reported issue, Lending and borrowing functionality is susceptible to front-running, is fixed.

We classify grief attacks as minor. Nevertheless, the consequences of this can be major because order book spoofers might create a large series of orders to signal a deep and liquid order book to cancel all the orders at once to create panic and cause excess volatility. This practice is considered a financial crime in many jurisdictions. Orderbook spoofing can be combined with other trades that exploit high volatility regimes to create excess returns. The infamous flash crash in 2010, resulting in a $1 trillion in market capitalization loss, was caused by a single non-institutional trader who created a series of spoofed orders and canceled them quickly.

Recommendation
We recommend implementing a logic that locks up the liquidity behind orders.

Status
Acknowledged

13. Suboptimal liquidation incentives put a higher burden on the insurance fund and lead to systemic risk

---

A loan subject to liquidation offers the largest incentive to the liquidator when the collateral ratio is just at the liquidation threshold and decays linearly with the payoff as the collateral ratio gets reduced until there is no incentive to liquidate when the collateral ratio is 100%.

This disincentivizes late liquidations, which are the most costly for the protocol and can significantly burden the protocol's risk fund

Recommendation
We recommend more stable or even increasing liquidation incentives as the collateral ratio diminishes.

Status
Acknowledged. The client states that they understand the limitations of the liquidation incentives and we will re-evaluate how to improve it in a future update.

14. Repayment fees charged during compensation potentially put borrowers at risk of liquidation

---

The `compensate` function in `Size.sol` allows a caller, who is both a borrower and a lender, to compensate outstanding debt with available credit in another loan. In addition to repaying the loan's debt, a repayment protocol fee is charged and paid in collateral tokens, effectively reducing the borrower's collateral-to-debt ratio and potentially causing the borrower to become liquidateable if the collateral ratio falls below the liquidation threshold.

As this issue only affects the caller, who is the borrower, and can not be exploited by an attacker to affect other arbitrary borrowers, we classify this issue as minor.

Recommendation
We recommend calling the `validateUserIsNotLiquidatable` function in the `compensate` function to ensure that the borrower's collateral ratio remains healthy.

Status
Resolved

15. Overdue loans moved to Aave's variable pool can be at risk of liquidation

---

An overdue loan is liquidated by moving the fixed-rate loan to Size's variable Aave pool via the `moveLoanToVariablePool` function in `VariableLibrary.sol`.

However, if the loan's assigned collateral, determined by the `getFOLAssignedCollateral` function, is only slightly above the healthy Aave LTV threshold, the loan will be put at risk of liquidation should the collateral value decrease, resulting in the borrower being charged a liquidation fee.

This issue only arises should Size's liquidation threshold configuration parameter, `crLiquidation`, be set to a value that differs from Aave's risk parameters.

Recommendation
We recommend checking if the resulting LTV of the moved loan is above Aave's risk parameter plus a margin of safety and if not, fall back to the regular liquidation process without moving the loan to the variable Aave pool.

Status
Resolved

16. Loans with a maturity date set to the current block timestamp can be immediately liquidated

---

The validation of the `lendAsMarketOrder` function's `LendAsMarketOrderParams` arguments, carried out by the `validateLendAsMarketOrder` function in `LendAsMarketOrder.sol`, ensures that the loan's due date is in the future by reverting if `params.dueDate < block.timestamp`.

However, the due date can be set to the current block timestamp. If the borrow offer's yield curve is configured to start at a maturity of `0`, it results in the loan being immediately considered overdue, i.e., `LoanStatus.OVERDUE`, by the `getLoanStatus` in `LoanLibrary.sol`. Consequently, the loan can be liquidated immediately. Additionally, having a small maturity date may be exploited by another borrower to early exit a soon-to-be overdue loan without being able to timely repay the loan, causing the loan to be liquidated.

Similarly, this issue is also present in the `validateBorrowAsMarketOrder` function in `BorrowAsMarketOrder.sol`.

Recommendation
We recommend implementing a protocol-defined minimum maturity for loans.

Status
Resolved

17. Overdue liquidation fee potentially exceeds the borrower's collateral balance, preventing the timely liquidation of the loan

---

The `executeLiquidateLoan` function in `LiquidateLoan.sol` liquidates an overdue but healthy fixed-rate loan by moving the loan to Size's variable Aave pool. The loan is moved by calling the `moveLoanToVariablePool` function in `_executeLiquidateLoanOverdue`. Any error that occurs during the loan transfer is caught (e.g., insufficient supply), and the regular liquidation logic, i.e., the liquidator repaying the loan's debt and receiving the pro-rata collateral as compensation, is executed.

However, if the charged `collateralOverdueTransferFee` fee, a fixed fee quoted in the collateral token (e.g., `szWETH`), exceeds the borrower's collateral, the token transfer in lines `113-115` will revert, causing the liquidation to fail. Consequently, the overdue loan can not be liquidated until the borrower's collateral increases, or the borrower can be liquidated due to the collateral ratio falling below the liquidation threshold.

This issue is likely to occur for small-sized loans, where the `collateralOverdueTransferFee` fee is a significant portion of the borrower's collateral or even exceeds the required collateral due to a lack of ensuring that the minimum loan size bound, `minimumCreditBorrowAToken`, is larger than the overdue transfer fee. Thus, we classify this issue as Minor.

Recommendation
We recommend either validating that the `minimumCreditBorrowAToken` configuration parameter, quoted in borrow tokens, is larger than the `collateralOverdueTransferFee` fee or

checking the borrower's collateral token balance to ensure sufficient tokens are available to pay the fee.

Status
Acknowledged. The client states that they understand the limitations of their liquidation path design when moving a loan to the variable pool and will re-evaluate how to improve it in a future update.

18. Size protocol admin can potentially liquidate all positions by setting crLiquidation higher than all borrower collateral ratios

In `UpdateConfig.sol` the `executeUpdateConfig` function can be called by a Size Protocol Admin to update a single config setting in the protocol.

However, it is possible to update config settings in such a way as to potentially be able to liquidate all positions. For example, the `crOpening` can be set higher than all borrower collateral ratios, followed by setting the `crLiquidation` threshold to a slightly higher number. As a result, all positions can be liquidated. For example, this can be achieved by calling the `executeUpdateConfig` function with `crOpening = 1000%` and subsequently calling `executeUpdateConfig` again with `crLiquidation = 999%`. Consequently, all loans can be liquidated.

Please note that this can only be performed by a permissioned operator with the `DEFAULT_ADMIN_ROLE` role granted.

Recommendation
We recommend validating the difference between the `crOpening` and `crLiquidation` on each update or setting a limit for `crOpening` and `crLiquidation`. Moreover, the wallet associated with the `DEFAULT_ADMIN_ROLE` role should be a multisig wallet to allow for multiple signatories to agree on a transaction before it is confirmed.

As an additional safety check, certain critical settings, such as `crOpening` and `crLiquidation` could be restricted from being changed only when the Size protocol is paused.

Status
Acknowledged. The client states that they prefer not to specify a ceiling for `crOpening` and `crLiquidation` because this issue would still apply (assuming the admin is trusted and is passing a bad value by mistake and not maliciously). For example, if a ceiling is set at 200%, the admin could still unintentionally liquidate all positions by raising the minimum collateral ratio, even if that is below the ceiling. The client intends to address this issue by ensuring the admin is a multisig wallet and by developing custom tools that allow signatories to verify changes before they are implemented.

19. Size protocol admin operator can potentially liquidate all positions by assigning a malicious oracle

In `UpdateConfig.sol` the `executeUpdateConfig` function can be called by a Size Protocol Admin to update a single config setting in the protocol.
However, it is possible to assign a malicious oracle to allow them (or an attacker) to drain the entire contract.

Recommendation
We recommend calling `executeUpdateConfig` should not be able to immediately reassign the oracle (or any other config for that matter), but rather give market participants adequate time to close their positions (in case they wish to) before any updated config protocol settings are assigned.

Additionally, we recommend giving market participants ample time to settle their positions before any other market parameters are modified.

Status
Acknowledged. The client states that the admin is trusted by assumption.

20. Withdraw event can emit an incorrect withdrawal amount

In `Withdraw.sol`, the emitted `Withdraw` event in the `executeWithdraw` function incorrectly uses `params.amount` as the `amount` value instead of the calculated `amount` value returned from `Math.min` in lines `47` or `52`.

This can lead to misleading `Withdraw` event logs, potentially causing external off-chain systems to consider a withdrawal of a certain amount has been made from the protocol when, in fact, it has not.

Recommendation

We recommend changing the `Withdraw` to emit the calculated `amount` value returned by the call to `Math.min` in line `47` or `52` instead.

Status

Resolved

21. `aaveOracle.getAssetPrice` calls ChainLink `AccessControlledOffchainAggregator.latestAnswer`, which is deprecated and not recommended to be used

In `VariablePoolPriceFeed.sol`, an instance of `AaveOracle` is used to get the price of an asset using the `getAssetPrice` function. However, the `aaveOracle.getAssetPrice` function calls ChainLink's `AccessControlledOffchainAggregator.latestAnswer,` which is deprecated and not recommended to be used.

Recommendation

We recommend using an Oracle implementation that calls `latestRoundData` instead.

Status
Resolved. The client chose to amend the AaveOracle (out of scope of this report) on their fork to use `latestRoundData` with staleness checks.

22. Lack of a secondary fallback price feed oracle

In `VariablePoolPriceFeed.sol` and `PriceFeed.sol`, the price returned by the `getPrice` function is from a ChainLink oracle. However, if the ChainLink oracle is reporting a zero or stale price, then the `getPrice` function will revert, and therefore, the Size protocol will be unusable during this time.

Recommendation
We recommend implementing a secondary oracle in case the underlying ChainLink oracle fails.

Status
Acknowledged. The client states that they have weighed the pros and cons of using a two-oracle system and are accepting the risk of not having a fallback Oracle. This is due to the shortcomings of managing integrations with third-party providers. In addition, the client points out that Aave v3 does not have a fallback Oracle, which can be found here: Aave V3 Oracle Contract on Etherscan.

21. Validate that the Oracle address is an IPriceFeed contract

In `Initialize.sol`, the `validateInitializeOracleParams` function validates the `IPriceFeed` interface for the `InitializeOracleParams.priceFeed` address. This ensures that the `PriceFeed` is not set to a zero address.

However, if the address is a non-zero address, it is possible to set the `IPriceFeed` to an incorrect contract instance.

Recommendation

We recommend further validating the `IPriceFeed` interface for `InitializeOracleParams.priceFeed` address correctness by calling `getPrice` on the interface, and if it reverts, the `priceFeed` address is not valid. This reduces the risk of deploying with a non-valid price feed oracle.

Status
Resolved

24. Claiming a loan that is already claimed returns a misleading `LOAN_NOT_REPAID` error

---

In `Claim.sol`, the `executeClaim` function allows a lender to claim the repayment of a loan with accrued interest from the Variable Pool. This moves the loan `LoanStatus.CLAIMED` state.

However, if the `executeClaim` function is called again for the same loan, then a `LOAN_NOT_REPAID` error is returned, which may be misleading for the lender of the protocol.

Recommendation
We recommend adding another validation in the `validateClaim` function to check the `state.getLoanStatus(loan) == LoanStatus.CLAIMED` to return a new custom error such as `Errors.LOAN_ALREADY_CLAIMED(params.loanId)`.

Status
Resolved

25. When a vault `aToken` balance is insufficient, the returned error is `PROXY_CALL_FAILED`, which is unclear to the user

---

In `VariableLibrary.sol`, the `transferBorrowAToken` is used to transfer `aTokens` from one user to another via a vault proxy pattern.

However, if a lender or borrower's `BorrowAToken` balance in their vault is insufficient to complete the `aToken` transfer, the error returned to the caller is `PROXY_CALL_FAILED,` which can lead to confusion to the protocol user.

Recommendation
We recommend the `transferBorrowAToken` function returns the underlying `ERC20` error from the vault proxy call.

Status
Resolved

26. It is not possible to cancel lend limit orders

In `Size.sol`, a lender can call `lendAsLimitOrder` to create a new limit order specifying a `maxDueDate` and a `YieldCurve`.

Given that a lend limit offer is considered `null` when the `maxDueDate` and the Yield Curve is empty (as defined in the `OfferLibrary.isNull` function), then it should be possible for a user to submit a transaction to zero out these values in their limit order.

However, the `LendAsLimitOrder.validateLendAsLimitOrder` function reverts if the user sets `maxDueDate` on the limit order to zero, thereby preventing the `LimitOrder` from being fully canceled from the order book.

Recommendation
We recommend allowing the user to set `maxDueDate` in a `LimitOrder` to zero to allow it to be canceled.

Status
Resolved

Informational Notes

27. In the validateCompensate function, the repaid state of the loan to compensate is checked twice

In `Compensate.sol,` the `validateCompensate` function checks whether the loan is repaid using `state.getLoanStatus(loanToCompensate) == LoanStatus.REPAID` and then in the next line checks the loan is not repaid using the same condition, `state.getLoanStatus(loanToCompensate) != LoanStatus.REPAID`, again.

Recommendation
We recommend removing the second condition as it can be safely removed since if the first check is `false` the loan is not repaid.

Status
Resolved

28. Add a fallback price discovery mechanism against Oracle manipulation

Oracles might fail or be manipulated. Consequently, no liquidity might back the price reported by the oracle. To ensure that there is liquidity behind the price signaled by the oracle, the collateral can be sold off in auctions or trusted pools, e.g., via an aggregator.

Recommendation
We recommend selling the collateral via a trusted mechanism instead of sending it to the liquidator to add a layer of protection against oracle attacks and malicious behavior of the oracle.

Status

Acknowledged.

29. Validate that configured addresses are distinct during initialization

---

The `validateInitializeDataParams` does not check if the configured borrow token, pool address, and collateral token are distinct addresses. Copy-paste errors by admins could lead to duplicates, which introduce a configuration risk.

Recommendation

We recommend validating that there are no duplicates in the submitted addresses.

Status

Acknowledged. The client states they will perform sanity checks with deployment script tests to mitigate this issue.

30. Validate that a borrow order is below the dust limit in the validate function

---

The `validateBorrowAsMarketOrder` function does not check if the borrow order is below the dust limit. This is inefficient as too late checks are costly.

Recommendation

We recommend that the `validateBorrowAsMarketOrder` validates that the order is above the dust limit.

Status
Acknowledged. The client states that for the sake of consistency with lendAsMarketOrder, which has a more complex "amount" calculation, they choose to not perform this validation in the beginning of the function and leave the implementation as is.

31. Liquidations revert if they are not profitable for the liquidator

The `validateMinimumCollateralProfit` reverts if the minimum liquidation profit is not met. This might happen if the price of the collateral asset drops quickly. However, this puts an additional burden on the protocol as all checks of a liquidation have already been performed, and further price drops might make liquidation more costly.

Recommendation
As a longer term measure we recommend implementing an auto-liquidation functionality that can be switched on by admins in an emergency state. This functionality could liquidate the asset on behalf of the protocol, if prices drop suddenly. Instead of admins it could be triggered by an oracle that detects high volatility of the base asset. Such a logic could carry out liquidations if the desired profit is not more than a percentage threshold (to prevent draining the emergency pool) and the collateral ratio is in a predefined range.

Status
Acknowledged. The client states that they understand the limitations of their liquidation incentives and that they will re-evaluate how to improve it in a future update.

32. USDC stablecoin centralization risk

USDC is a centralized stablecoin that can be controlled by operators outside of the Size protocol. For example, the operators of the USDC stablecoin can blacklist the Size protocol, thereby inadvertently freezing the Size protocol USDC liquidity. This can be achieved by the USDC operator calling the `blacklist` function with the Size Proxy contract address.

Recommendation

We recommend, if possible, using a decentralized stablecoin as the lending token.

Status
Acknowledged

33. Avoid using the latest Solidity compiler version to reduce compiler bug risks.

The Size contract uses the `pragma solidity 0.8.24` statement. This sets the version of the solidity compiler, `solc`, to `0.8.24`, which is currently the latest version. However, occasionally, new compiler versions can introduce bugs and unknown vulnerabilities, and therefore it may pose a risk to use the latest compiler version.

Recommendation
We recommend avoiding using the very latest compiler version - especially if not using any of the latest compiler features. This is to reduce the risk of potentially introducing unknown compiler bugs. Perhaps consider using `solc` version `0.8.23`.

Status
Resolved

34. In Size.sol, the State struct is imported twice

In `Size.sol`, the `State` struct is imported twice, in line `39` and in line `46`.

Recommendation
We recommend that the second import of `State` on line `46` is removed.

Status
Resolved

35. In Errors.sol, the @title attribute is incorrectly marked as “Events”

In `Errors.sol`, the `@title` attribute is incorrectly marked as *“Events”*.

Recommendation
We recommend updating the `@title` attribute to *“Errors”*.

Status
Resolved

36. All possible event parameters should be indexed

In `Events.sol` all Size protocol events are defined. However, some of the events are not indexing all of the possible parameters.

An indexed event field makes the field more quickly accessible to off-chain tools that parse events. Some events defined in the `Events.sol` do not index all possible fields.

Recommendation
We recommend that each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Status
Resolved

37. AccountingLibrary.chargeRepayFee could be renamed or split into two functions

In `AccountingLibrary.sol`, the `chargeRepayFee` function performs two distinct actions, which the function name does not infer. To prevent any confusion, this function could be renamed or split into two separate functions, which are to charge fees and reduce the loan's `issuanceValue`.

Recommendation
We recommend renaming `chargeRepayFee` to `chargeRepayFeeAndReduceIssuanceValue` or moving the adjustment of the `issuanceValue` to a new internal function called `reduceIssuanceValue`.
This would clearly separate the accounting into two steps: charge fees and reduce principal.

Status
Resolved

38. `RiskLibrary.getMinimumCollateralOpening` function is never used in the protocol

In `RiskLibrary.sol`, the `getMinimumCollateralOpening` function is not called anywhere in the protocol.

Recommendation
We recommend removing the `RiskLibrary.getMinimumCollateralOpening` function definition.

Status
Resolved

39. `UserLibrary` definition is empty and can be removed

In `UserLibrary.sol`, the `UserLibrary` library definition is empty.

Recommendation

Consider removing the `UserLibrary` library definition in the `UserLibrary.sol` file.

Status
Resolved

40. The White Paper states that rates can be negative while the protocol defines YieldCurve.rates values as unsigned integers

The Size White Paper states that rates can be negative. However, in `YieldCurveLibrary.sol`, the `YieldCurve` struct `rates` type is an unsigned integer (`uint256[]`).

Recommendation
We recommend changing the Size White Paper or updating `YieldCurve` struct `rates` type to an `int256` (with consideration to test this change).

Status
Resolved

41. Empty test definitions

There are some test files that include empty test definitions.
Some examples are `test_LiquidateLoan_liquidateLoan_move_to_VP_fails_if_VP_does_not_have_enough_liquidity` and `test_LiquidateLoan_liquidateLoan_charge_repayFee` which are defined in `test/LiquidateLoan.t.sol`.

Recommendation
We recommend implementing all tests.

Status
Acknowledged

42. Typo in `VariableLibrary.depositUnderlyingBorrowTokenToVariablePool` in `@dev` docs

In `VariableLibrary.sol`, there is a NatSpec docs typo for the function `depositUnderlyingBorrowTokenToVariablePool`. The docs mention *"from has approvet to"*.

Recommendation
Change “approvet” to “approved”.

Status
Resolved

43. In the White Paper, a borrower can specify `N>=1` LLOs while the protocol only allows a single LLO to be specified

The Size White Paper specifies that the borrower can set `N>=1` lending limit orders (LLOs). However, the `BorrowAsMarketOrderParams` only allow to specify a *single* lender which can only have a single `loanOffer` attached.

Recommendation
We recommend updating the Size White Paper or the protocol to reflect the desired functionality.

Status
Acknowledged

44. Empty function definition for `validateUpdateConfig`

in `UpdateConfig.sol`, the `validateUpdateConfig` function is empty and thus not needed. The call to this function from the `updateConfig` function in `Size.sol` should also be removed or commented out.

Recommendation
We recommend removing the function definition and its invocation or adding update config validation logic as suggested by the function name.

Status
Acknowledged

45. Miscellaneous Comments

The following are suggestions to improve the overall code quality and readability.

- Assure consistency in code comments if validations should not happen either never mark them with `N/A` or always mark them with `N/A`
    - Rename the constants to include both units and values e.g., `PERCENT` to `100_PERCENT` or `ONE_PERCENT`
- Follow the Checks-Effects-Interactions pattern such that the code is in line with your documentation
    - Swap the lines `libraries/fixed/actions/LiquidateLoan.sol:96` and `97`
    - Swap the lines `libraries/fixed/actions/Claim.sol:42` and `43`
- Write function-level documentation and document significant internal functions
- Write integration tests to ensure that one action does not impact other actions. For example, in `Compensate.t.sol`, the `compensate` functionality is tested. However, claiming immediately following compensation is not tested. If a test for the claim functionality following `compensate` was included in the same test, then the issue Compensating a loan results in the lender being unable to claim and the borrower unable to repay and exit the loan would have been identified as part of the test suite.
- Add a `getPrice` function to `SizeView.sol` that returns the current Oracle price, as this may be useful for clients.
- Remove slither statements in

- `libraries/variable/VariablePool.sol:80,100`
- `libraries/fixed/actions/Compensate.sol:77`
- `libraries/variable/VariableLibrary.sol:147`
- `libraries/fixed/AccountingLibrary.sol:80`

Appendix: Test Cases

1. Test case for "Inability to liquidate compensated and receivable loans"

```
diff --git a/test/fixed/Compensate.t.sol b/test/fixed/Compensate.t.sol
index cef1eb5..be5bc16 100644
--- a/test/fixed/Compensate.t.sol
+++ b/test/fixed/Compensate.t.sol
@@ -9,6 +9,41 @@ import {Loan, LoanLibrary} from
"@src/libraries/fixed/LoanLibrary.sol";
 contract CompensateTest is BaseTest {
     using LoanLibrary for Loan;

+    function
test_Compensate_compensate_compensated_loan_cannot_be_liquidated() public {
+        _deposit(alice, weth, 100e18);
+        _deposit(alice, usdc, 100e6);
+        _deposit(bob, weth, 100e18);
+        _deposit(bob, usdc, 100e6);
+        _deposit(james, weth, 100e18);
+        _deposit(james, usdc, 100e6);
+        _lendAsLimitOrder(alice, 12, 1e18, 12);
+        _lendAsLimitOrder(bob, 12, 1e18, 12);
+        _lendAsLimitOrder(james, 12, 1e18, 12);
+        uint256 loanToCompensateId = _borrowAsMarketOrder(bob /** @note
borrower */, alice /** @note lender */, 20e6, 12);
+        uint256 loanToRepay = _borrowAsMarketOrder(alice /** @note
borrower */, james /** @note lender */, 20e6, 12);
+        uint256 repayFee = size.repayFee(loanToCompensateId);
+
+        uint256 repaidLoanDebtBefore = size.getDebt(loanToRepay);
+        uint256 compensatedLoanCreditBefore =
size.getCredit(loanToCompensateId);
+
+        _compensate(alice, loanToRepay, loanToCompensateId);
+
+        uint256 repaidLoanDebtAfter = size.getDebt(loanToRepay);
+        uint256 compensatedLoanCreditAfter =
```

```
size.getCredit(loanToCompensateId);
+
+        assertEq(repaidLoanDebtAfter, repaidLoanDebtBefore - 2 * 20e6 -
repayFee);
+        assertEq(compensatedLoanCreditAfter, compensatedLoanCreditBefore -
2 * 20e6);
+        assertEq(
+            repaidLoanDebtBefore - repaidLoanDebtAfter - repayFee,
+            compensatedLoanCreditBefore - compensatedLoanCreditAfter
+        );
+
+        _setPrice(0.1e18);
+
+        assertTrue(size.isUserLiquidatable(bob));
+        assertFalse(size.isLoanLiquidatable(loanToCompensateId)); //
@audit-info Loan is not liquidateable
+    }
+
     function
test_Compensate_compensate_reduces_repaid_loan_debt_and_compensated_loan_cr
edit() public {
        _deposit(alice, weth, 100e18);
        _deposit(alice, usdc, 100e6);
```

2. Test case for "Second-order loan (SOL) self-liquidation is malfunctioning due to using the wrong borrower to determine the collateral ratio"

```
diff --git a/test/fixed/SelfLiquidateLoan.t.sol
b/test/fixed/SelfLiquidateLoan.t.sol
index 8dffd89..c5a40e7 100644
--- a/test/fixed/SelfLiquidateLoan.t.sol
+++ b/test/fixed/SelfLiquidateLoan.t.sol
@@ -10,6 +10,44 @@ import {Vars} from "@test/BaseTestGeneral.sol";
 contract SelfLiquidateLoanTest is BaseTest {
     using LoanLibrary for Loan;
```

```
+    function test_SelfLiquidateLoan_selfliquidateLoan_SOL_doesnt_work()
public {
+        _setPrice(1e18);
+        _updateConfig("repayFeeAPR", 0);
+
+        _deposit(alice, weth, 150e18);
+        _deposit(alice, usdc, 100e6 + size.config().earlyLenderExitFee);
+        _deposit(bob, weth, 150e18);
+        _deposit(candy, weth, 150e18);
+        _deposit(candy, usdc, 100e6 + size.config().earlyLenderExitFee);
+        _deposit(james, usdc, 100e6);
+        _deposit(liquidator, usdc, 10_000e6);
+
+        assertEq(size.collateralRatio(bob), type(uint256).max);
+
+        _lendAsLimitOrder(alice, 12, 0, 12);
+        _lendAsLimitOrder(candy, 12, 0, 12);
+        _lendAsLimitOrder(james, 12, 0, 12);
+
+        uint256 folId = _borrowAsMarketOrder(alice /** @note borrower */,
candy /** @note lender  */, 100e6, 12);
+        uint256 solId = _borrowAsMarketOrder(candy /** @note borrower */,
james /** @note lender  */, 30e6, 12, [folId]);
+
+        assertEq(size.getFOLAssignedCollateral(folId), 150e18);
+        assertEq(size.getDebt(folId), 100e6);
+        assertEq(size.getCredit(folId), 70e6);
+        assertEq(size.collateralRatio(alice), 1.5e18);
+        assertTrue(!size.isUserLiquidatable(bob));
+        assertTrue(!size.isLoanLiquidatable(folId));
+
+        _setPrice(0.5e18);
+        assertEq(size.collateralRatio(alice), 0.75e18);
+
+        Vars memory _before = _state();
+
+        _selfLiquidateLoan(candy, folId);
+        vm.expectRevert();
+        _selfLiquidateLoan(james, solId);
```

```
+     }
+
      function
test_SelfLiquidateLoan_selfliquidateLoan_rapays_with_collateral() public {
          _setPrice(1e18);
          _updateConfig("repayFeeAPR", 0);
```

3. Test case for "Self-liquidating loans round up the burned repay fee, resulting in potentially insufficient debt tokens for subsequent SOL self-liquidations"

```
diff --git a/src/libraries/fixed/RiskLibrary.sol
b/src/libraries/fixed/RiskLibrary.sol
index f1217c6..ca3aea5 100644
--- a/src/libraries/fixed/RiskLibrary.sol
+++ b/src/libraries/fixed/RiskLibrary.sol
@@ -43,9 +43,10 @@ library RiskLibrary {
      function isLoanSelfLiquidatable(State storage state, uint256 loanId)
public view returns (bool) {
          Loan storage loan = state.data.loans[loanId];
          LoanStatus status = state.getLoanStatus(loan);
+         Loan storage fol = state.getFOL(loan);
          // both FOLs and SOLs can be self liquidated
          return
-             (isUserLiquidatable(state, loan.generic.borrower) &&
status.either([LoanStatus.ACTIVE, LoanStatus.OVERDUE]));
+             (isUserLiquidatable(state, fol.generic.borrower) &&
status.either([LoanStatus.ACTIVE, LoanStatus.OVERDUE]));
      }

      function isLoanLiquidatable(State storage state, uint256 loanId)
public view returns (bool) {
diff --git a/test/fixed/SelfLiquidateLoan.t.sol
b/test/fixed/SelfLiquidateLoan.t.sol
index 8dffd89..08c838d 100644
--- a/test/fixed/SelfLiquidateLoan.t.sol
+++ b/test/fixed/SelfLiquidateLoan.t.sol
@@ -10,6 +10,41 @@ import {Vars} from "@test/BaseTestGeneral.sol";
```

```
 contract SelfLiquidateLoanTest is BaseTest {
     using LoanLibrary for Loan;

+    function
test_SelfLiquidateLoan_selfliquidateLoan_SOL_insufficient_debt_token_repay_
fee() public {
+        _setPrice(1e18);
+        // @audit-info Default repay fee 0.005e18 as configured in
`Deploy.sol`
+
+        _deposit(alice, weth, 200e18);
+        _deposit(alice, usdc, 100e6 + size.config().earlyLenderExitFee);
+        _deposit(bob, weth, 200e18);
+        _deposit(candy, weth, 200e18);
+        _deposit(candy, usdc, 100e6 + size.config().earlyLenderExitFee);
+        _deposit(james, usdc, 100e6);
+        _deposit(liquidator, usdc, 10_000e6);
+
+        assertEq(size.collateralRatio(bob), type(uint256).max);
+
+        _lendAsLimitOrder(alice, 12, 0, 12);
+        _lendAsLimitOrder(candy, 12, 0, 12);
+        _lendAsLimitOrder(james, 12, 0, 12);
+
+        uint256 folId = _borrowAsMarketOrder(alice /** @note borrower */,
candy /** @note lender  */, 100e6, 12);
+        uint256 solId = _borrowAsMarketOrder(candy /** @note borrower */,
james /** @note lender  */, 30e6, 12, [folId]);
+
+        assertTrue(!size.isLoanLiquidatable(folId));
+
+        _setPrice(0.5e18);
+
+        Vars memory _before = _state();
+
+        _selfLiquidateLoan(candy, folId);
+
+        assertTrue(size.isUserLiquidatable(alice));
+
```

```
+         vm.expectRevert();
+         _selfLiquidateLoan(james, solId);
+     }
+
      function
test_SelfLiquidateLoan_selfliquidateLoan_rapays_with_collateral() public {
          _setPrice(1e18);
          _updateConfig("repayFeeAPR", 0);
```

4. Test case for "Borrower loses entire collateral if loan is liquidated when overdue"

```
diff --git a/test/fixed/LiquidateLoan.t.sol
b/test/fixed/LiquidateLoan.t.sol
index 41daf39..e06786c 100644
--- a/test/fixed/LiquidateLoan.t.sol
+++ b/test/fixed/LiquidateLoan.t.sol
@@ -294,6 +294,30 @@ contract LiquidateLoanTest is BaseTest {
          } catch {}
      }

+     function
test_LiquidateLoan_overdue_loan_moves_entire_collateral_to_variable_pool()
public {
+         _deposit(alice, usdc, 1000e6);
+         _deposit(bob, weth, 500e18);
+         uint256 amount = 800e6;
+         _lendAsLimitOrder(alice, 12, 0.03e18, 12);
+         uint256 loanId = _borrowAsMarketOrder(bob, alice, amount, 12);
+         Loan memory loan = size.getLoan(loanId);
+
+         // move time forward to allow for liquidation due to being overdue
+         vm.warp(block.timestamp + 12);
+         LoanStatus s = size.getLoanStatus(loanId);
+         // loan should be overdue and liquidatable
+         assertTrue(s == LoanStatus.OVERDUE);
+         assertTrue(size.isLoanLiquidatable(loanId));
+
```

```
+         Vars memory _before = _state();
+         // bob collateral should be intact (500 WETH)
+         assertEq(_before.bob.collateralAmount, 500e18);
+         _liquidateLoan(james, loanId, 1e17);
+         Vars memory _after = _state();
+         // bob collateral is 0 (all moved to Variable Pool)
+         assertEq(_after.bob.collateralAmount, 0);
+     }
+
```

Disclaimer

Oak Security GmbH