Term Structure Labs Security Review

Reviewed by: ZeroK

Content

1. About ZeroK

I’m a Web3 security researcher with two years of experience, specializing in Solidity, Sway, EVMs, FuelVM, and DeFi protocols. I began my bug hunting journey on Immunefi in 2024 and achieved Elite rank within just 8 months. In 2025, I was recognized as an All-Star member on Immunefi for my significant contributions and high-impact vulnerability discoveries.

2. Disclaimer

The details shared in this report are for informational purposes only and are not in intended to encourage or discourage users or investors from engaging with the mentioned bug bounty program. This report highlights a vulnerabilities i discovered while reviewing the specified protocol during a set period in specifi time and repository, Please conduct your own research and due diligence before investing in or working on mentioned protocol.

3. About Term Structure Labs (TSL)

Term Structure Institutional (TSI) is an institutional-grade platform enabling clients to borrow and lend digital assets at fixed rates within a secure Fireblocks multi-party computation (MPC) wallet environment and a reliable TSI Electronic Communication Network (ECN). By combining transparent and flexible cryptocurrency financing with robust security and institutional-grade features, TSI empowers institutions, lenders, borrowers, and traders to participate in fixed-income markets with greater efficiency and confidence.

For more information check this Link

4. Risk Classification

All impacts and severity levels outlined in this report follow Immunefi's official Impact and severity classification standards which be found Here

5. Security Assessment Summary

This audit was conducted as part of an Immunefi Invite-Only Program (IOP) focused on the **TSI protocol's core contracts**, specifically Settlement.sol and LoanLib. The IOP was initially scheduled for 7 days and later extended to 14 days. During this period, I performed an in-depth review of the protocol's logic, with a strong emphasis on identifying critical vulnerabilities and improving overall contract security.

5.1. Protocol Summary

Project name	Term Structure labs
Repository	TSI-contracts
Type of project	lend and borrow, defi
Audit timeline	9 days
Review commit hash	f0dd14d1370b460765441b1611c6355c4ed6c77c
Fixes review commit hash	23c947137597cc22a6409aa61f69ce687235097d

5.2. Scope

File	nSLOC
src/settlement.sol	270 nSloc
src/lib/loanLib.sol	308 nSloc
Total	**578 nSloc**

6. Finding

6.1. Critical Findings

[C-01] malicious borrower can take theft of other borrower collateral

Description

The createSettlement function allows a single lender to create loan positions for multiple borrowers, each with different debt, borrowing amounts, and collateral amounts. Additionally, there are functions that enable either the borrower or the lender to modify these loan positions, such as adding, removing, or repaying collateral or debt. However, many of these functions are callable even after the settlement has expired or before the loan has been marked as settled. A malicious borrower could exploit this design to withdraw other borrowers' collateral from the lender by leveraging the **'addCollateralBeforeSettleand'** **'repay'** functions.

Vulnerability Details

The function createSettlement implemented as below, in our case taker type == lend:

```
function createSettlement(
      string memory _settlementId,
      SettleInfo calldata _settleInfo,
      string[] memory _loanIds,
      LoanInfo[] calldata _loans,
      bytes calldata _signature
  ) external nonReentrant {
      // // Verify signature
      bytes32 digest = _getSettlementHash(_settlementId, _settleInfo, _loanIds, _loans);
      if (!SignatureChecker.isValidSignatureNow(_operator, digest, _signature)) {
          revert InvalidSignature();
      }
      bytes32 settlementId = _settlementId.toBytes32();
      if (settlements[settlementId].taker != address(0)) {
          revert SettlementAlreadyExists(settlementId);
      }
      if (_settleInfo.expiryTime < block.timestamp) {
          revert SettlementExpired(block.timestamp, _settleInfo.expiryTime);
      }
      if (_settleInfo.taker != msg.sender) {
          revert TakerNotMatched(settlementId, _settleInfo.taker, msg.sender);
      }
      if (_loans.length == 0) {
          revert EmptyLoan();
      }
      uint256 totalCollateralAmt;
      uint256 totalDebtAmt;
      address debtToken = _loans[0].debtTokenAddr;
      address collateralToken = _loans[0].collateralTokenAddr;
      for (uint256 i = 0; i < _loanIds.length; i++) {
      bytes32 loanId = _loanIds[i].toBytes32();
      LoanInfo memory loan = _loans[i];
          if (loans[loanId].maker != address(0)) {
         revert LoanAlreadyExisted(loanId);
          }
          loan.settlementId = settlementId;
```

```
 loan.settled = false;
 loans[loanId] = loan;
 totalCollateralAmt += loan.debtData.collateralAmt;
 totalDebtAmt += loan.debtData.debtAmt;
 if (_settleInfo.takerType == TakerType.BORROW && loan.collateralTokenAddr !=
collateralToken) {
 revert TakerNotMatched(loanId, collateralToken, loan.collateralTokenAddr);
 } else if (_settleInfo.takerType == TakerType.LEND && loan.debtTokenAddr !=
debtToken) {
 revert TakerNotMatched(loanId, debtToken, loan.debtTokenAddr);
   }
 }
 if (_settleInfo.takerType == TakerType.LEND) {
    uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
    if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
        revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE
        * totalCollateralAmt);
    }
    allowance = IERC20(debtToken).allowance(msg.sender, address(this));
    if (allowance < totalDebtAmt) {
        revert TokenAllowanceInsufficient(allowance, totalDebtAmt);
    }
 } else {
     uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, totalCollateralAmt);
     }
 }

 settlements[settlementId] = _settleInfo;

 emit SettlementCreated(msg.sender, settlementId);
}
```

The lender (who is the msg.sender) is required to approve the Settlement contract to spend an amount equal to 10 times the total collateral received from all borrowers. This requirement ensures that the liquidation process can be executed smoothly when needed. For example, if the total collateral from 5 loan positions is 5,000 ETH (where each borrower provides 1,000 ETH as collateral to receive 1.9 million USDC), the lender must approve the Settlement contract to spend 10 times that amount (e.g 50,000 ETH)

```
uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this))
 if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
 revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE *
totalCollateralAmt);
```

Imagine that there are 6 loans created 5 honest users(borrower), and 1 malicious borrower. Let’s say the createSettlement function was invoked correctly, and everything initially worked as expected. Now, the malicious borrower can exploit this system to steal collateral from the lender that originally came from honest borrowers.

- The malicious borrower can simply wait until the loan position expires (settlement expires). Meanwhile, the other honest borrowers would execute calls to the settle function, transferring their collateral to the lender's wallet.

- The borrower invokes the **'addCollateralBeforeSettle'** function, and increase the collateral amount to equal to 5k eth which is same collateral value that borrower transfer to lender in settle function call(borrower can increase this value up to all collateral lender hold and gave approve to settlement.sol contract):

```
function addCollateralBeforeSettle(string memory _loanId, uint256 addedAmount) public
nonReentrant {
bytes32 loanId = _loanId.toBytes32();

LoanInfo memory loanInfo = loans[loanId];
if (loanInfo.maker == address(0)) {
    revert LoanNotFound(loanId);
}
if (loanInfo.settled) {
    revert LoanAlreadySettled(loanId);
}

loanInfo.addCollateral(addedAmount);
if (settlements[loanInfo.settlementId].takerType == TakerType.BORROW) {
    uint256 allowance = IERC20(loanInfo.collateralTokenAddr).allowance(msg.sender,
    address(this));
    if (allowance < loanInfo.debtData.collateralAmt) {
        revert TokenAllowanceInsufficient(allowance, loanInfo.debtData.collateralAmt);
   }
}

loans[loanId] = loanInfo;

emit CollateralAdded(msg.sender, loanId, addedAmount);
}
```

- we can see the addCollateralBeforeSettle is callable even if the settlement is expired(loan expired) and it increase the collateral debt without transfer the added collateral.

- After that, the borrower can call the repay function. What this function does is transfer the debt amount of the loan 1.9 million USDC in this case, to the lender, and then transfer the collateral back to the borrower. The collateral should be 1,000 ETH, and it should only be transferred after the settlement is executed. However, in this scenario, the borrower would actually receive 5,000 ETH, which the lender got from other borrowers, This is possible due to a few reasons:

A) The lender has already approved the contract to spend collateral on their behalf.

B) The borrower increased the loan's collateral or debt, and although the loan expired, it can still be modified and repaid.

C) The health check will pass because the collateral amount was increased this allows the malicious user to effectively steal the lender's collateral and also impact the liquidation process, creating a serious vulnerability.

Impact Details

due to lack of check in addCollateralBeforeSettle and repay functions, malicious user can take theft of the lender collateral.

Recommend

It’s better to prevent any calls to the repay function when settled == false or when the loan has expired. This is because the addCollateralBeforeSettle function can be invoked before settled becomes true, allowing an attacker to increase the collateral amount of a position. If this happens, the attacker should then invoke the settle function, which actually takes the collateral from them. Otherwise, the repay function will revert because of the added settled == true check (the same check can also be applied to the liquidation process). Regarding the removeCollateral function, although it’s possible to call it, it would revert in most cases since it checks the position’s health. When the collateral is zero, the ltv typically exceeds the mltv (maximum Liquidation Threshold) under normal conditions, preventing collateral removal.

6.1. Critical Findings

[C-02] direct theft of users funds when expired loan get liquidated

Description

Currently, the **Settlement.sol** contract allows either the lender or the borrower to create a settlement by invoking the **createSettlement** function. This function is designed to work as intended only if the operator properly validates all input parameters. When creating a settlement, if the caller is the lender (indicated by takerType == lender), then only the borrower or maker can execute the settlement. This design ensures that both parties remain aligned and that the agreement is mutually acknowledged. However, the borrower can choose to not agree to the settlement once it's created and simply let it expire. There is a significant incentive for the borrower to do this. By refusing the settlement, the borrower can effectively exploit the system to seize the lender's collateral. This collateral might originally come from another honest borrower, plus any liquidation incentives, and the borrower can even reclaim their own collateral if they fully repay the debt. At first glance, this might appear fair since the lender still receives the repaid debt amount. However, as explained in the Vulnerability Details section, this setup can be manipulated, leading to unfair outcomes and potential losses for the lender.

Vulnerability Details

First thing, imagine a settlement agreed on by operator between alice(malicious borrower) and bob(victim lender) created by invoking the createSettlement function, lender or taker which is bob created the settlement:

```
function createSettlement(
    string memory _settlementId,
    SettleInfo calldata _settleInfo,
    string[] memory _loanIds,
    LoanInfo[] calldata _loans,
    bytes calldata _signature
  ) external nonReentrant {
      //PART 1: Verify signature
      bytes32 digest = _getSettlementHash(_settlementId, _settleInfo, _loanIds, _loans);
      if (!SignatureChecker.isValidSignatureNow(_operator, digest, _signature)) {
          revert InvalidSignature();
     }
     //PART 2: Create sanity checks
     bytes32 settlementId = _settlementId.toBytes32();
     if (settlements[settlementId].taker != address(0)) {
         revert SettlementAlreadyExists(settlementId);
     } // if the settlement already exists, revert
     if (_settleInfo.expiryTime < block.timestamp) {
         revert SettlementExpired(block.timestamp, _settleInfo.expiryTime);
     } // if the settlement is in past(it can be equal)
     if (_settleInfo.taker != msg.sender) {
         revert TakerNotMatched(settlementId, _settleInfo.taker, msg.sender);
     } // taker must be the caller
     if (_loans.length == 0) {
         revert EmptyLoan();
     }
```

```
// PART 3: Create settlement and loans
uint256 totalCollateralAmt;
uint256 totalDebtAmt;
address debtToken = _loans[0].debtTokenAddr; // debt token addr for first loan(USDC)
address collateralToken = _loans[0].collateralTokenAddr; // collateral token addr for first
loan(ETH)

 for (uint256 i = 0; i < _loanIds.length; i++) {
      bytes32 loanId = _loanIds[i].toBytes32();
      LoanInfo memory loan = _loans[i];
      if (loans[loanId].maker != address(0)) {
          revert LoanAlreadyExisted(loanId);
     } // if the loan created before, revert

     loan.settlementId = settlementId; //set the settlement id for each loan
     loan.settled = false;
     loans[loanId] = loan; //UPDATE storage
     totalCollateralAmt += loan.debtData.collateralAmt;
     totalDebtAmt += loan.debtData.debtAmt;

     if (_settleInfo.takerType == TakerType.BORROW && loan.collateralTokenAddr !=
     collateralToken) {
         revert TakerNotMatched(loanId, collateralToken, loan.collateralTokenAddr);

    } //if the taker is borrowing then check if the all loan have same collateral token
    else if (_settleInfo.takerType == TakerType.LEND && loan.debtTokenAddr !=
    debtToken) {
        revert TakerNotMatched(loanId, debtToken, loan.debtTokenAddr);
    } //if taker is lending then check if the debt token is same for all loans
 }
 if (_settleInfo.takerType == TakerType.LEND) {
    uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE *
         totalCollateralAmt);
 } // lender must approve this contract before doing anything
```

```
    allowance = IERC20(debtToken).allowance(msg.sender, address(this));
    if (allowance < totalDebtAmt) {
        revert TokenAllowanceInsufficient(allowance, totalDebtAmt);
    } // lender must approve this contract before doing anything(for debt token)
 } else {
     uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, totalCollateralAmt);
     }
  }

  settlements[settlementId] = _settleInfo;

  emit SettlementCreated(msg.sender, settlementId);
}
```

We can see that Bob (the msg.sender, acting as the taker/lender) grants full approval to the Settlement contract to manage the total collateral amount he has received from a later settlement call by the borrower. For example, let's say the collateral amount is 5,000 ETH when 1 ETH equals 2,000 USDC. Bob now waits for Alice (the borrower) to invoke the settle function to complete the settlement, update the settlement status to Settled, and transfer the collateral while receiving the debt amount. However, Alice doesn't call the settle function; instead, she waits for the settlement to expire (which happens in 1 day in our example). Rather than settli

It’s important to note that no money or collateral is actually transferred between Alice and Bob until the settle function is invoked. This function is the only way to finalize the settlement, as shown in the implementation below

```
function settle(string memory _loanId) public nonReentrant {
bytes32 loanId = _loanId.toBytes32();
LoanInfo memory loanInfo = loans[loanId]; // get the loan info data

if (loanInfo.maker == address(0)) {
    revert LoanNotFound(loanId);
}
if (loanInfo.maker != msg.sender) {
    revert MakerNotMatched(loanInfo.settlementId, loanId, loanInfo.maker, msg.sender);
} //only maker can invoke settle
if (loanInfo.settled) {
    revert LoanAlreadySettled(loanId);
}

SettleInfo memory settleInfo = settlements[loanInfo.settlementId];
if (settleInfo.expiryTime < block.timestamp) {
    revert SettlementExpired(block.timestamp, settleInfo.expiryTime);
}

if (settleInfo.takerType == TakerType.BORROW) {
    if (loanInfo.lender != loanInfo.maker) {
        revert LenderNotMatched(loanId, loanInfo.lender, loanInfo.maker);
} //lender == msg.sender
if (loanInfo.borrower != settleInfo.taker) {
    revert BorrowerNotMatched(loanId, loanInfo.borrower, settleInfo.taker);
} // borrower == taker
uint256 allowance = IERC20(loanInfo.collateralTokenAddr).allowance(msg.sender,
 address(this)); // msg.sender is maker
```

```
   if (allowance < Constants.ALLOWANCE_SCALE * loanInfo.debtData.collateralAmt) {
       revert TokenAllowanceInsufficient(
           allowance, Constants.ALLOWANCE_SCALE * loanInfo.debtData.collateralAmt
       );
   }
 } else {
   if (loanInfo.borrower != loanInfo.maker) {
       revert BorrowerNotMatched(loanId, loanInfo.borrower, loanInfo.maker);
   } // borrower == msg.sender
   if (loanInfo.lender != settleInfo.taker) {
       revert LenderNotMatched(loanId, loanInfo.lender, settleInfo.taker);
   } // lender == taker
  }
  loanInfo.checkHealth(_oracle);

  loans[loanId].settled = true;

  IERC20(loanInfo.debtTokenAddr).safeTransferFrom(
      loanInfo.lender, loanInfo.borrower, loanInfo.debtData.borrowedAmt
  );
  IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(
      loanInfo.borrower, loanInfo.lender, loanInfo.debtData.collateralAmt
  );
  IERC20(loanInfo.debtTokenAddr).safeTransferFrom(loanInfo.lender, _feeCollector,
 loanInfo.debtData.feeAmt);

  emit Settled(msg.sender, loanId);
 }
```

now after 10 days, maturity passed, and before this, bob got another settlement for the same amount with eve as borrower, bob now hold 5000 eth, and eve got 9M USDC, now alice invoke liquidity function with the loanID equal to the expired one:

```
function liquidate(string memory _loanId, uint256 liquidationAmt) external nonReentrant {
  bytes32 loanId = _loanId.toBytes32();
  LoanInfo memory loanInfo = loans[loanId];
  (uint256 collateralToLiquidator, uint256 collateralToProtocol) =
     loanInfo.liquidate(loanId, _oracle, _minimumDebtValue, liquidationAmt);

  IERC20(loanInfo.debtTokenAddr).safeTransferFrom(msg.sender, loanInfo.lender,
  liquidationAmt);
  IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(loanInfo.lender, msg.sender,
  collateralToLiquidator);
  IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(loanInfo.lender, _feeCollector,
  collateralToProtocol);

  if (loanInfo.debtData.debtAmt == 0) {
     IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(
        loanInfo.lender, loanInfo.borrower, loanInfo.debtData.collateralAmt
     );
     delete loans[loanId];
  } else {
     loans[loanId] = loanInfo;
  }
  emit Liquidated(
    loanId,
    msg.sender,
    liquidationAmt,
    collateralToLiquidator,
    collateralToProtocol,
    loanInfo.debtData.collateralAmt
  );
}
```

```
function liquidate(
    LoanInfo memory loanInfo,
    bytes32 loanId,
    IOracle oracle,
    uint256 minimumDebtValue,
    uint256 amount
) internal view returns (uint256 collateralToLiquidator, uint256 collateralToProtocol) {
    (bool liquidateable,, uint256 maxLiquidationAmt, PriceInfo memory collateralPrice,
     PriceInfo memory debtPrice) = liquidaitonInfo(loanInfo, oracle, minimumDebtValue);
    if (!liquidateable) {
        revert SettlementErrors.CannotLiquidate(loanId);
    }
    if (
        amount > loanInfo.debtData.debtAmt
           || amount > maxLiquidationAmt * (Constants.DECIMAL_BASE +
           Constants.MAX_LTV_BUFFER) / Constants.DECIMAL_BAS
   ) {
      revert SettlementErrors.LiquidationAmtExceedsMax(amount, maxLiquidationAmt);
   }

   uint256 ltvBefore = LoanLib.calculateLtv(loanInfo, collateralPrice, debtPrice);
   (uint256 pdToPc, uint256 denominator) = _calculatePaToPbUpround(debtPrice,
    collateralPrice);
    uint256 removedCollateralAmt =
       amount * pdToPc * collateralPrice.tokenDenominator / (denominator *
       debtPrice.tokenDenominator)

    if (removedCollateralAmt > loanInfo.debtData.collateralAmt) {
        revert SettlementErrors.CollateralRemovedAmtExceedsCollateralAmt(
            loanInfo.debtData.collateralAmt, removedCollateralAmt
        );
   }
  collateralToLiquidator =
      removedCollateralAmt + (removedCollateralAmt *
     Constants.REWARD_TO_LIQUIDATOR) / Constants.DECIMAL_BASE;
  collateralToProtocol = (removedCollateralAmt * Constants.REWARD_TO_PROTOCOL)
  / Constants.DECIMAL_BASE
  if (collateralToLiquidator >= loanInfo.debtData.collateralAmt) {
      collateralToLiquidator = loanInfo.debtData.collateralAmt;
      collateralToProtocol = 0;

 } else if (collateralToLiquidator + collateralToProtocol >= loanInfo.debtData.collateralAmt)
{
    collateralToProtocol = loanInfo.debtData.collateralAmt - collateralToLiquidator;
 }
 loanInfo.debtData.collateralAmt =
    loanInfo.debtData.collateralAmt - collateralToLiquidator - collateralToProtocol;
 loanInfo.debtData.debtAmt -= amount;

 uint256 ltvAfter = calculateLtv(loanInfo, collateralPrice, debtPrice);
 if (ltvAfter == 0) {
     return (collateralToLiquidator, collateralToProtocol);
 }
 if (ltvAfter >= ltvBefore || ltvAfter < loanInfo.debtData.mltv –
Constants.MAX_LTV_BUFFER) {
     revert SettlementErrors.LtvInvalidAfterLiquidation(ltvAfter, ltvBefore);
 }
}

 function liquidaitonInfo(LoanInfo memory loanInfo, IOracle oracle, uint256
 minimumDebtValue)
    internal
    view
     returns (
        bool liquidateable, // is liquidateable
        bool deliverable, // is deliverable
        uint256 maxLiquidationAmt,
        PriceInfo memory collateralPrice,
        PriceInfo memory debtPrice
    )
 {
```

```
(collateralPrice, debtPrice) = getPriceInfos(loanInfo, oracle);
uint256 ltv = LoanLib.calculateLtv(loanInfo, collateralPrice, debtPrice);
 if (loanInfo.debtData.maturity <= block.timestamp) { // if maturity reached
     if (ltv >= Constants.DECIMAL_BASE) {
          deliverable = true;
     } else {
         liquidateable = true;
         maxLiquidationAmt = loanInfo.debtData.debtAmt; // for mature, set all debt amount
     } //even if collateral is much more than debt, it is liquidateable because its mature
 } else {
     if (ltv >= Constants.DECIMAL_BASE) {
          deliverable = true;
     } else if (ltv >= loanInfo.debtData.lltv) {
         liquidateable = true;
         maxLiquidationAmt = _calculateMaxLiquidationAmt(loanInfo, collateralPrice,
         debtPrice);
         uint256 remainningDebtAmt = loanInfo.debtData.debtAmt - maxLiquidationAmt;
         if (remainningDebtAmt > 0 && remainningDebtAmt < _
         calculateMinimumDebtAmt(minimumDebtValue, debtPrice))
         {
             maxLiquidationAmt = loanInfo.debtData.debtAmt;
          }
        }
    }
}
```

so what is the benefits for alice to do this since she's forced to pay all debt to lender? alice will get more eth or collateral than the debt she pays plus she affect bob new loan Position and prevent the loan position for eve to be completed normally and lead to liquidation which might revert, so what alice get is:

- first she invoke the liquidate with: loanID = loan id between him and bob, and liquidationAmt = full repay the debt amount which is 9.5M USDC

- the internal liquidate function invoked, it first check if the position is matured in line below:

```
  if (loanInfo.debtData.maturity <= block.timestamp) { // if maturity reached
else {
           liquidateable = true;
           maxLiquidationAmt = loanInfo.debtData.debtAmt; // for mature, set all debt amount
       } //even if collateral is much more than debt, it is liquidateable because its mature
```

- the max liquidation is the full amount of the debt or 9.5M USDC, then the internal liquidate function calculate the eth that the liquidator should get in return (alice or the borrower in our case) in line below:

```
uint256 removedCollateralAmt =
    amount * pdToPc * collateralPrice.tokenDenominator / (denominator *
    debtPrice.tokenDenominator)
```

- she gets around 4750 in total of 5k eth in collateral in bob address, and plus she gets the liquidation incentive in line below:

```
collateralToLiquidator =
    removedCollateralAmt + (removedCollateralAmt *
    Constants.REWARD_TO_LIQUIDATOR) / Constants.DECIMAL_BASE;
 collateralToProtocol = (removedCollateralAmt *
Constants.REWARD_TO_PROTOCOL) / Constants.DECIMAL_BASE;
```

- the incentive can be around another 30 eth and 40 eth for protocol as fee(nearly value), and debtAmt become 0(since we payed fully) and then after that the liquidate function will invoke transfer actions as below:

```
IERC20(loanInfo.debtTokenAddr).safeTransferFrom(msg.sender, loanInfo.lender,
liquidationAmt);
IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(loanInfo.lender, msg.sender,
collateralToLiquidator);
IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(loanInfo.lender, _feeCollector,
collateralToProtocol);

if (loanInfo.debtData.debtAmt == 0) {
    IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(
        loanInfo.lender, loanInfo.borrower, loanInfo.debtData.collateralAmt
    );
    delete loans[loanId];
```

- alice transfer 9.5 million to bob(flashloan can be used without taking fees), alice who is msg.sender gets around 4800 eth(9.6M in USDC) and the protocol gets the fee(50 eth) and any remaining collateral will be sent to borrower because debt is 0, which is alice herself. in total alice got around +1M USDC profit.

- not only this, this action affect the agreement(the loan position) between bob and eve, because when eve want to payback the debt to get her collateral, the repay function will revert, because bob does not hold the collateral to pay eve back when repay function invoked, even same can be true for liquidation, in this case eve can not get back her collateral until bob get collateral to his wallet, liquidation process revoked by bots and reverts, it might get delievered which lead to same result, reverting, in this case the whole protocol can be affected by this root cause.

this took from this example in TSI docs with full liquidation : https://docs.institutional.ts.finance/features/liquidation-and-physical-delivery#liquidation-penalty-calculation

Impact Details

expired loan never removed or deleted which can be used to take theft of lender funds

Recommend

simple fix is as below:

- if the loan expired, instead of reverting when settle called, delete the loan, even if the caller is not the maker.

- allow bot only to invoke the liquidate function, one address can be created and act like a bot by checking events emitting.

6.2. High Findings

[H-01] any users with expired loan(not settled) can take theft of lenders collateral when the collateral price increase

Description

The removeCollateral function is designed to allow the borrower to reduce their collateral in a loan position, typically when the collateral value increases and exceeds the debt amount. For example, if the collateral is initially 100 ETH (worth 2 million USDC) against a debt of 1.9 million USDC, and the ETH price rises to $3,000 per ETH (making the collateral worth 3 million USDC), the borrower can remove excess collateral to bring it closer to the required 2 million USDC. However, a malicious borrower could exploit this feature to steal the lender's collateral. This is possible because the removeCollateral function allows calls even for expired or non settled positions. If the position is not settled, the collateral has not yet been transferred by borrower to lender, and the borrower can still call removeCollateral to withdraw free collateral from the lender's wallet, since the lender had already approved the settlement contract to manage the collateral on their behalf.

Vulnerability Details

When a loan position is created, it can either be a lend position or a borrow position. In our case, it's a lend position, which means the caller of the createSettlement function is the lender (the taker).

```
function createSettlement(
  string memory _settlementId,
  SettleInfo calldata _settleInfo,
  string[] memory _loanIds,
  LoanInfo[] calldata _loans,
  bytes calldata _signature
) external nonReentrant {
   // // Verify signature
   bytes32 digest = _getSettlementHash(_settlementId, _settleInfo, _loanIds, _loans);
   if (!SignatureChecker.isValidSignatureNow(_operator, digest, _signature)) {
       revert InvalidSignature();
   }
   bytes32 settlementId = _settlementId.toBytes32();
   if (settlements[settlementId].taker != address(0)) {
       revert SettlementAlreadyExists(settlementId);
   }
   if (_settleInfo.expiryTime < block.timestamp) {
      revert SettlementExpired(block.timestamp, _settleInfo.expiryTime);
   }
   if (_settleInfo.taker != msg.sender) {
       revert TakerNotMatched(settlementId, _settleInfo.taker, msg.sender);
   }
   if (_loans.length == 0) {
       revert EmptyLoan();
   }
   uint256 totalCollateralAmt;
   uint256 totalDebtAmt;
   address debtToken = _loans[0].debtTokenAddr;
   address collateralToken = _loans[0].collateralTokenAddr;
   for (uint256 i = 0; i < _loanIds.length; i++) {
       bytes32 loanId = _loanIds[i].toBytes32();
       LoanInfo memory loan = _loans[i];
       if (loans[loanId].maker != address(0)) {
           revert LoanAlreadyExisted(loanId);
       }
  loan.settlementId = settlementId;
  loan.settled = false;
  loans[loanId] = loan;
  totalCollateralAmt += loan.debtData.collateralAmt;
  totalDebtAmt += loan.debtData.debtAmt;
  if (_settleInfo.takerType == TakerType.BORROW && loan.collateralTokenAddr !=
  collateralToken) {
      revert TakerNotMatched(loanId, collateralToken, loan.collateralTokenAddr);
  } else if (_settleInfo.takerType == TakerType.LEND && loan.debtTokenAddr !=
  debtToken) {
       revert TakerNotMatched(loanId, debtToken, loan.debtTokenAddr);
  }
}
 if (_settleInfo.takerType == TakerType.LEND) {
     uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE *
         totalCollateralAmt);
 }

  allowance = IERC20(debtToken).allowance(msg.sender, address(this));
  if (allowance < totalDebtAmt) {
      revert TokenAllowanceInsufficient(allowance, totalDebtAmt);
 }
} else {
    uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, totalCollateralAmt);
     }
 }
 settlements[settlementId] = _settleInfo;

 emit SettlementCreated(msg.sender, settlementId);
}
```

```
 function addCollateralBeforeSettle(string memory _loanId, uint256 addedAmount) public
nonReentrant {
    bytes32 loanId = _loanId.toBytes32();

    LoanInfo memory loanInfo = loans[loanId];
    if (loanInfo.maker == address(0)) {
        revert LoanNotFound(loanId);
    }
    if (loanInfo.settled) {
        revert LoanAlreadySettled(loanId);
    }
     loanInfo.addCollateral(addedAmount);
     if (settlements[loanInfo.settlementId].takerType == TakerType.BORROW) {
         uint256 allowance = IERC20(loanInfo.collateralTokenAddr).allowance(msg.sender,
          address(this));
          if (allowance < loanInfo.debtData.collateralAmt) {
              revert TokenAllowanceInsufficient(allowance, loanInfo.debtData.collateralAmt);
          }
     }

     loans[loanId] = loanInfo;

     emit CollateralAdded(msg.sender, loanId, addedAmount);
 }
```

When the position type is set to lend, the lender is required to grant approval to the settlement contract to transfer collateral on their behalf. After that, the lender must wait for the borrower to invoke the settle function to complete the settlement process. However, a borrower can bypass this intended flow by skipping the settle function entirely and instead invoking the removeCollateral function. this exploit works as follows: the borrower waits for the collateral price to increase significantly (for example, from $2,000 to $3,000 per ETH), then calls **removeCollateral** to withdraw collateral, the removeCollateral function does not check whether the position is settled or expired meaning it can be invoked even on expired or non settled positions. As a result, the borrower can withdraw the collateral without repaying any debt, stealing the lender’s funds.

```
function removeCollateral(string memory _loanId, uint256 removeCollateralAmt) external
nonReentrant {
    bytes32 loanId = _loanId.toBytes32();
    LoanInfo memory loanInfo = loans[loanId];
    loanInfo.checkBorrower(loanId, msg.sender);
    loanInfo.removeCollateral(removeCollateralAmt, _oracle);

    loans[loanId] = loanInfo;

    IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(loanInfo.lender, loanInfo.borrower,
    removeCollateralAmt);

    emit CollateralRemoved(loanId, removeCollateralAmt);
}

function removeCollateral(LoanInfo memory loanInfo, uint256 removingCollateralAmt, IOracle
oracle) internal view {
  if (loanInfo.debtData.collateralAmt < removingCollateralAmt) {
      revert SettlementErrors.CollateralRemovedAmtExceedsCollateralAmt(
          loanInfo.debtData.collateralAmt, removingCollateralAmt
      );
  }
  loanInfo.debtData.collateralAmt -= removingCollateralAmt;
  checkHealth(loanInfo, oracle);
}
```

Impact Details

Any borrower can effectively steal yield or collateral from any position created where the maker is the user (borrower). By waiting for the collateral price to rise, the borrower can invoke functions like removeCollateral even on expired or non settled positions, thereby extracting collateral without repaying the debt. This allows a malicious borrower to profit from the increased collateral value and siphon funds from the lender’s address.

Recommend

It is recommended to add a check to prevent functions like removeCollateral from being invoked on non settled or expired positions. these functions should include a check ensuring that settled == true(recommended) and that the position has not expired (i.e., expired > block.timestamp). This would prevent borrowers from exploiting these functions to extract collateral without repaying the debt and mitigate the risk of collateral theft from lenders.

6.2. High Findings

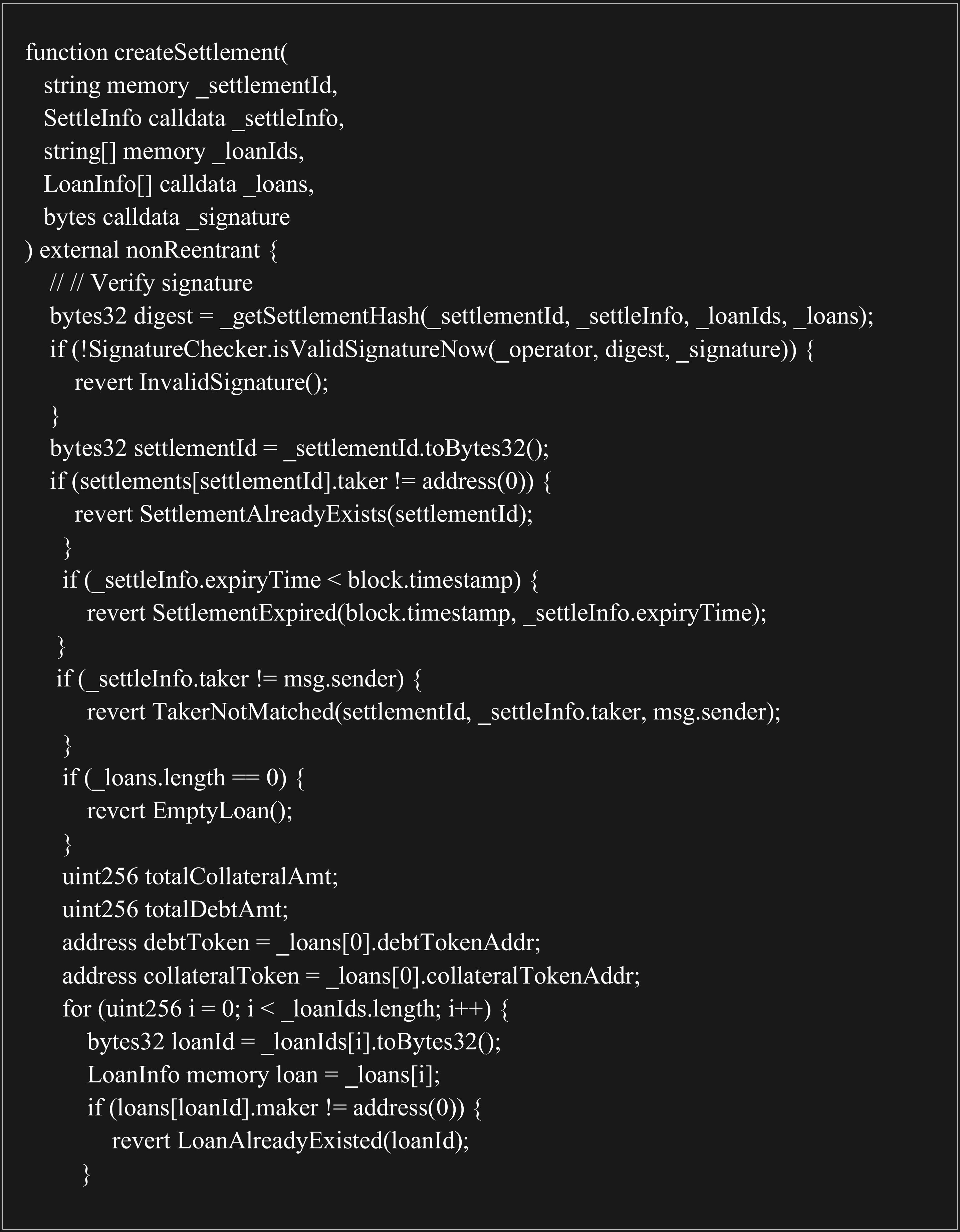
[H-02] settlement functionality can be break forever and blocking settlement actions

Description

The function **createSettlement** allow users to create settlement, either lend type or borrow type, if its lend type then the borrower(maker) should invoke settle function, if borrow type then lender(maker) should invoke settle function, in both cases, both borrower and lender should give approve to settlement contract to spent tokens on behalf of them, this is required to complete settlement and to meet the TSI wallet implementation requirement(1 key with fireblocks, another one with caller, no action invoked if one of them not sign specific transaction). however there is another function that can be used by malicious user to break the whole settlement mechanism and prevent the protocol to work as expected, which is the addCollateralBeforeSettle, this function implemented to allow the caller to increase the collateralAmt for any loan position without actually transfer collateral to the contract or the borrower address, this open possibilities to malicious user to invoke this function to increase collateralAmt to high value that exceed borrower collateral amount that hold prevent any call to settlement actions no matter if the settlement type is lend or borrow

Vulnerability Details

the function createSettlement implemented as below, it checks approval for both sides, borrower or lender or both, depending on the settle type:



```
function createSettlement(
   string memory _settlementId,
   SettleInfo calldata _settleInfo,
   string[] memory _loanIds,
   LoanInfo[] calldata _loans,
   bytes calldata _signature
) external nonReentrant {
    // // Verify signature
    bytes32 digest = _getSettlementHash(_settlementId, _settleInfo, _loanIds, _loans);
    if (!SignatureChecker.isValidSignatureNow(_operator, digest, _signature)) {
        revert InvalidSignature();
    }
    bytes32 settlementId = _settlementId.toBytes32();
    if (settlements[settlementId].taker != address(0)) {
        revert SettlementAlreadyExists(settlementId);
      }
      if (_settleInfo.expiryTime < block.timestamp) {
          revert SettlementExpired(block.timestamp, _settleInfo.expiryTime);
     }
     if (_settleInfo.taker != msg.sender) {
          revert TakerNotMatched(settlementId, _settleInfo.taker, msg.sender);
      }
      if (_loans.length == 0) {
          revert EmptyLoan();
      }
      uint256 totalCollateralAmt;
      uint256 totalDebtAmt;
      address debtToken = _loans[0].debtTokenAddr;
      address collateralToken = _loans[0].collateralTokenAddr;
      for (uint256 i = 0; i < _loanIds.length; i++) {
          bytes32 loanId = _loanIds[i].toBytes32();
          LoanInfo memory loan = _loans[i];
          if (loans[loanId].maker != address(0)) {
              revert LoanAlreadyExisted(loanId);
          }
    loan.settlementId = settlementId;
    loan.settled = false;
    loans[loanId] = loan;
    totalCollateralAmt += loan.debtData.collateralAmt;
    totalDebtAmt += loan.debtData.debtAmt;
    if (_settleInfo.takerType == TakerType.BORROW && loan.collateralTokenAddr !=
    collateralToken) {
        revert TakerNotMatched(loanId, collateralToken, loan.collateralTokenAddr);
    } else if (_settleInfo.takerType == TakerType.LEND && loan.debtTokenAddr !=
    debtToken) {
                revert TakerNotMatched(loanId, debtToken, loan.debtTokenAddr);
    }
  }
  if (_settleInfo.takerType == TakerType.LEND) {
     uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
         revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE *
         totalCollateralAmt);
    }
    allowance = IERC20(debtToken).allowance(msg.sender, address(this));
    if (allowance < totalDebtAmt) {
        revert TokenAllowanceInsufficient(allowance, totalDebtAmt);
    }
  } else {
      uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
      if (allowance < totalCollateralAmt) {
          revert TokenAllowanceInsufficient(allowance, totalCollateralAmt);
     }
  }

  settlements[settlementId] = _settleInfo;

  emit SettlementCreated(msg.sender, settlementId);
}
```

required check for allowance done correctly since if its lend type, caller who is lender must approved settlement contract to spent collateral amount plus the debt, and same for borrower in this case, the problem arises when the **'addCollateralBeforeSettle'** invoked:

```
function addCollateralBeforeSettle(string memory _loanId, uint256 addedAmount) public
nonReentrant {
      bytes32 loanId = _loanId.toBytes32();

      LoanInfo memory loanInfo = loans[loanId];
      if (loanInfo.maker == address(0)) {
          revert LoanNotFound(loanId);
      }
      if (loanInfo.settled) {
          revert LoanAlreadySettled(loanId);
      }
      loanInfo.addCollateral(addedAmount);
      if (settlements[loanInfo.settlementId].takerType == TakerType.BORROW) {
          uint256 allowance = IERC20(loanInfo.collateralTokenAddr).allowance(msg.sender,
          address(this));
          if (allowance < loanInfo.debtData.collateralAmt) {
              revert TokenAllowanceInsufficient(allowance, loanInfo.debtData.collateralAmt);
          }
     }

      loans[loanId] = loanInfo;

      emit CollateralAdded(msg.sender, loanId, addedAmount);
 }
```

this function implemented in a way, that allow the borrower to increase the collateral before settle invoked, sometime borrower need to do so because the price of collateral might drop and lead to checkHealth revert when settle invoked, however, this function is invokable by anyone, anyone can call this function for any loan they want, this lead to malicious user to invoke the attack below:

- settlement between bob and eve created, bob approve 1000 eth to settlement contract to send it to eve, and eve approved to transfers 2M worth of USDC to bob when settle invoked

- alice recognized this, and invoke call to **addCollateralBeforeSettle** function, setting the addedAmount == uint(256).max -1 or any higher value than bob hold, alice will approve the contract to spend this value but she can remove the invoke after any call to addCollateralBeforeSettle, this is only required when the settle type is borrow, alice actually does not lose any value to break the whole settlement contract.

- the transferfrom revert since the borrower does not hold that much of collateral amount even if increase the allowance, this will affect users and TSI protocol since it break the core function of the protocol, and it can lead to protocol insolvency since it prevent any fee or reward to be sent to the protocol or the fee collector address the addCollateralBeforeSettle implemented in a way that it work well only when it get called by the loan maker address, why is that? because:

- the maker is the lender or borrower, this is forced since settle function force this rule and you can't set address different from both

- if the maker is borrower, then its meaningless for borrower to add collateral to a loan position without holding enough collateral, just to break the settlement action, this have no effects for both sides or even for the protocol

- if the maker is lender, then the maker can add more collateral, this to get more collateral to its address when the borrower address hold more collateral than the amount provided for current loan(borrower hold 2k eth but want to use 1k eth for this loan), but since the collateral get locked in lender address, there is no direct theft of the collateral(i still review the incentive for the lender to do so and increase collateral amount.

Impact Details

malicious user can break the core settlement functionality by invoking the addCollateralBeforeSettle function for unsettled loans

Recommend

i believe the below steps can be highly effective to prevent bugs like the one explained in this report:

- first thing allow only the borrower to invoke this function, its meaningless for lender to increase collateral for a position since the borrower should pay the collateral, if the collateral is less than the debt, then the borrower should take care of it since the borrower funds get liquidated

- allow invoking repay function only when loan.settled == true, this way we are sure the borrower paid collateral and transfer it to lender before repay, preventing any other attacks

- add expiry check for both repay and addCollateral

6.3. Medium Findings

[M-01] Any call to the repay function can potentially be front-run by a malicious actor, lead to prevent users from repaying on time

Description

The **Settlement#repay** function is designed to allow a borrower to repay the debt associated with a specific loan id, preventing liquidation. However, since the function does not have affected check for who can call it, any user including a malicious actor can front-run the borrower’s intended repayment. This means an attacker or even a lender could call the repay function just before the borrower does, effectively preventing the borrower from repaying in time and cause lose of gas for the borrower(griefing) In some cases, this could lead to the borrower’s position being liquidated despite their intention to repay

Vulnerability Details

the function repay implemented as below:

```
function repay(string memory _loanId, uint256 repayAmt, uint256 removeCollateralAmt)
external nonReentrant {
  bytes32 loanId = _loanId.toBytes32();
  LoanInfo memory loanInfo = loans[loanId];

  loanInfo.repay(repayAmt);
  if (removeCollateralAmt > 0) {
     loanInfo.checkBorrower(loanId, msg.sender);
     loanInfo.removeCollateral(removeCollateralAmt, _oracle);
  }

  IERC20(loanInfo.debtTokenAddr).safeTransferFrom(msg.sender, loanInfo.lender,
  repayAmt);
  if (removeCollateralAmt > 0) {
     IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(
        loanInfo.lender, loanInfo.borrower, removeCollateralAmt
    );
  }

  if (loanInfo.debtData.debtAmt == 0 && loanInfo.debtData.collateralAmt == 0) {
     delete loans[loanId];
  } else {
    loans[loanId] = loanInfo;
  }
  emit Repaid(loanId, repayAmt, removeCollateralAmt);
}
```

We can see that the **Settlement#repay** function allows any caller to set all the required inputs, regardless of who they are This can be useful because it means anyone can repay the borrower's debtsuch as a borrower's second address or even third parties

However, the function has an issue when the borrower themselves intends to repay all of their debt. This is because of how the **loanInfo.repay**function is implemented. Let's take a closer look at how this function works:

```
function repay(LoanInfo memory loanInfo, uint256 repayAmt) internal pure {
   if (loanInfo.debtData.debtAmt < repayAmt) {
       revert SettlementErrors.RepayAmountExceedsDebtAmt(loanInfo.debtData.debtAmt,
       repayAmt);
  }
  loanInfo.debtData.debtAmt -= repayAmt;
}
```

we can see that the check **if (loanInfo.debtData.debtAmt < repayAmt)** will revert if the loand.debtAmt is smaller than the repayAmt input, this can lead to scenario below to be possible:

- bob want to repay all of his debt back, which is 100 USDC
- alice(can be the lender too to get its value back when transfer invoked) recognize bob transaction and front run his call by repay back 1 usdc and setting all required input with setting the **removeCollateralAmt == 0** to avoid checking for caller is borrrower

- this way alice prevented bob to pay back all of his debt amount because the check in **if (loanInfo.debtData.debtAmt < repayAmt)** which equal to **99 < 100**

the issue exist in how the loanInfo.repay function work which it should set the repay amount to loanInfo.debtData.debtAmt when its bigger than the debt amount info

Impact Details

malicious user can front-run honest borrower who intend to pay their debt fully and cause damage to them and in certain cases can lead to liquidation

Recommend

a simple fix can be implemented as below:

```
function repay(LoanInfo memory loanInfo, uint256 repayAmt) internal pure {
  if (loanInfo.debtData.debtAmt < repayAmt) {
     repayAmt = loanInfo.debtData.debtAmt; //@audit since 100 is bigger than the debt
     amount, set repay amount to 99 and transfer back the 1 back to caller or borrower in repa
     function
  }

  loanInfo.debtData.debtAmt -= repayAmt;
}
```

6.4. Low Findings

[L-01] Any position can be closed (by repaying the debt) even after the maturity date has passed

Description

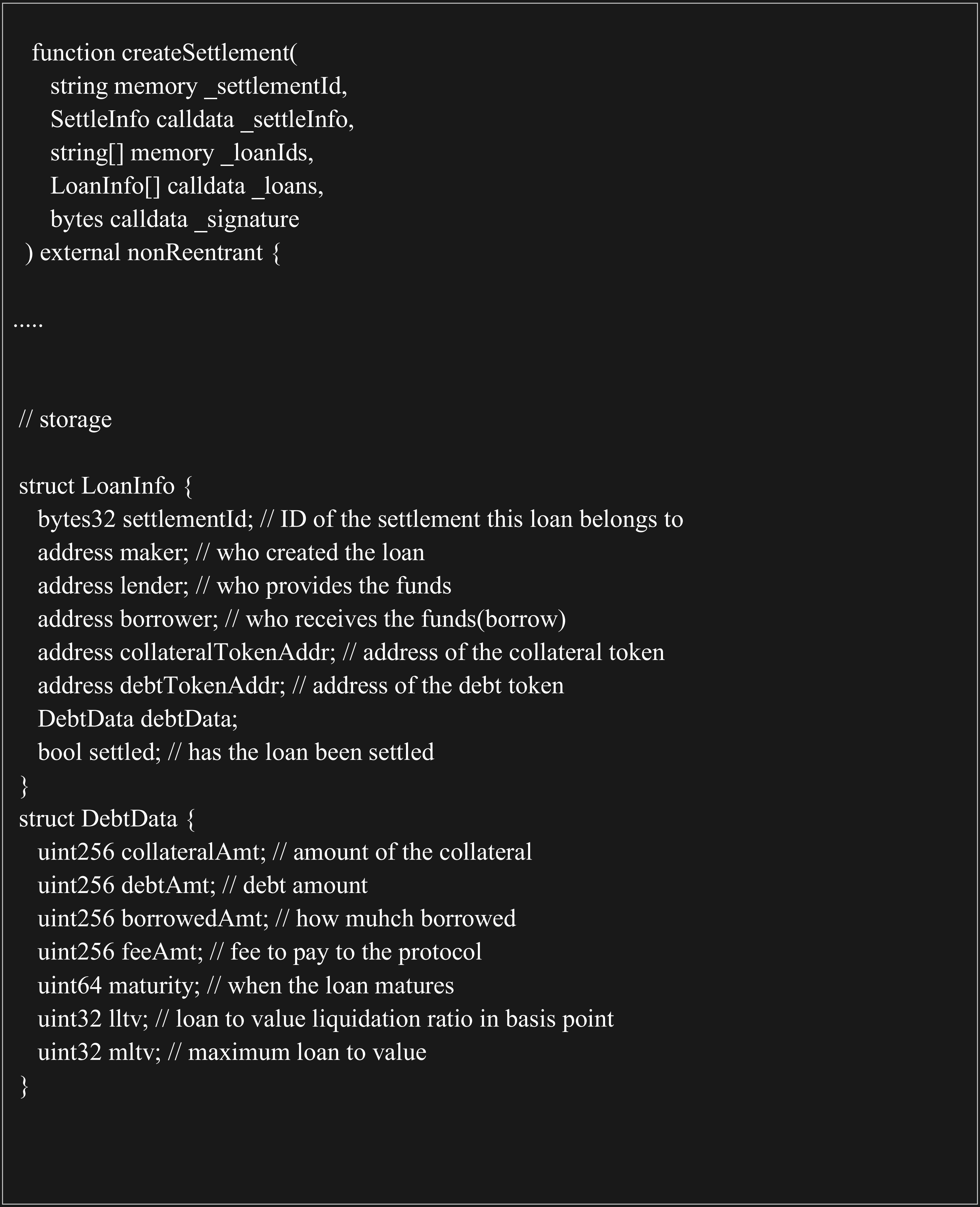
Any loan created has a maturity date, which is important because it enforces a repayment deadline. If the borrower does not repay on time, the position should be liquidated to ensure that the lender and the protocol are compensated. However, currently there is no restriction preventing repayment after the maturity date. Borrowers can repay or modify their position at any time, even after maturity. This undermines the liquidation process because users can front-run any attempt at liquidation by repaying their debt after maturity, effectively blocking liquidation for that loan position.

position should be liquidated right when the maturity reached for position:

https://docs.institutional.ts.finance/features/liquidation-and-physical-delivery#liquidation-ltv-lltv

Vulnerability Details

we can see when a loan created, the maturity time should be set as input parameter:



```
   function createSettlement(
      string memory _settlementId,
      SettleInfo calldata _settleInfo,
      string[] memory _loanIds,
      LoanInfo[] calldata _loans,
      bytes calldata _signature
  ) external nonReentrant {

.....


 // storage

 struct LoanInfo {
    bytes32 settlementId; // ID of the settlement this loan belongs to
    address maker; // who created the loan
    address lender; // who provides the funds
    address borrower; // who receives the funds(borrow)
    address collateralTokenAddr; // address of the collateral token
    address debtTokenAddr; // address of the debt token
    DebtData debtData;
    bool settled; // has the loan been settled
 }
 struct DebtData {
    uint256 collateralAmt; // amount of the collateral
    uint256 debtAmt; // debt amount
    uint256 borrowedAmt; // how muhch borrowed
    uint256 feeAmt; // fee to pay to the protocol
    uint64 maturity; // when the loan matures
    uint32 lltv; // loan to value liquidation ratio in basis point
    uint32 mltv; // maximum loan to value
 }
```

however, there is nothing prevent the borrower to invoke the repay function, even when the maturity passed, since there is not check to prevent invoking the function when timestamp > maturity:

```
function repay(string memory _loanId, uint256 repayAmt, uint256 removeCollateralAmt)
external nonReentrant {
    bytes32 loanId = _loanId.toBytes32();
    LoanInfo memory loanInfo = loans[loanId];

    loanInfo.repay(repayAmt);
    if (removeCollateralAmt > 0) {
        loanInfo.checkBorrower(loanId, msg.sender);
        loanInfo.removeCollateral(removeCollateralAmt, _oracle);
    }

    IERC20(loanInfo.debtTokenAddr).safeTransferFrom(msg.sender, loanInfo.lender,
    repayAmt);
    if (removeCollateralAmt > 0) {
        IERC20(loanInfo.collateralTokenAddr).safeTransferFrom(
            loanInfo.lender, loanInfo.borrower, removeCollateralAmt
        );
    }

    if (loanInfo.debtData.debtAmt == 0 && loanInfo.debtData.collateralAmt == 0) {
        delete loans[loanId];
    } else {
        loans[loanId] = loanInfo;
    }
    emit Repaid(loanId, repayAmt, removeCollateralAmt);
 }
```

this can lead calls to liquidation for mature position to fail since the loan deleted:

```
function liquidate(string memory _loanId, uint256 liquidationAmt) external nonReentrant {
  bytes32 loanId = _loanId.toBytes32();
  LoanInfo memory loanInfo = loans[loanId]; //@audit not exist
  (uint256 collateralToLiquidator, uint256 collateralToProtocol) =
      loanInfo.liquidate(loanId, _oracle, _minimumDebtValue, liquidationAmt);
```

this can block the liquidation process invoked by the bot, it can affect badly when batch of liquidation executed in one transaction

Impact Details

maturity of the loans not respected in repay function which lead to prevent liquidations

Recommend

add check for maturity in repay function, prevent repaying when the position passed the maturity

6.5. Insight Findings

[I-01] some checks should be added even if the operator checks each input parameters

Description

the createSettlement, allow a caller(the taker) to create a settlement, however there is some checks that its better to be added even if the operator checks each input before signing it as the team mentioned, these check can be for the feeAmount to be higher than zero, and the takerAmt is not zero, the maturity is more than the expiry, the maturity or expiry are not zero, the loanIDs and loans length are equal

Vulnerability Details

there is no Vulnerability, just best security practice in the function below:

```
function createSettlement(
    string memory _settlementId,
    SettleInfo calldata _settleInfo,
    string[] memory _loanIds,
    LoanInfo[] calldata _loans,
    bytes calldata _signature
  ) external nonReentrant {
    // // Verify signature
    bytes32 digest = _getSettlementHash(_settlementId, _settleInfo, _loanIds, _loans);
    if (!SignatureChecker.isValidSignatureNow(_operator, digest, _signature)) {
      revert InvalidSignature();
    }
    bytes32 settlementId = _settlementId.toBytes32();
    if (settlements[settlementId].taker != address(0)) {
      revert SettlementAlreadyExists(settlementId);
    }
    if (_settleInfo.expiryTime < block.timestamp) {
      revert SettlementExpired(block.timestamp, _settleInfo.expiryTime);
    }
    if (_settleInfo.taker != msg.sender) {
      revert TakerNotMatched(settlementId, _settleInfo.taker, msg.sender);
    }
    if (_loans.length == 0) {
      revert EmptyLoan();
    }
    uint256 totalCollateralAmt;
    uint256 totalDebtAmt;
    address debtToken = _loans[0].debtTokenAddr;
    address collateralToken = _loans[0].collateralTokenAddr;
    for (uint256 i = 0; i < _loanIds.length; i++) {
      bytes32 loanId = _loanIds[i].toBytes32();
      LoanInfo memory loan = _loans[i];
      if (loans[loanId].maker != address(0)) {
        revert LoanAlreadyExisted(loanId);
      }
  loan.settlementId = settlementId;
  loan.settled = false;
  loans[loanId] = loan;
  totalCollateralAmt += loan.debtData.collateralAmt;
  totalDebtAmt += loan.debtData.debtAmt;
  if (_settleInfo.takerType == TakerType.BORROW && loan.collateralTokenAddr !=
  collateralToken) {
     revert TakerNotMatched(loanId, collateralToken, loan.collateralTokenAddr);
  } else if (_settleInfo.takerType == TakerType.LEND && loan.debtTokenAddr !=
  debtToken) {
      revert TakerNotMatched(loanId, debtToken, loan.debtTokenAddr);
  }
 }
 if (_settleInfo.takerType == TakerType.LEND) {
    uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
    if (allowance < Constants.ALLOWANCE_SCALE * totalCollateralAmt) {
       revert TokenAllowanceInsufficient(allowance, Constants.ALLOWANCE_SCALE *
       totalCollateralAmt);
    }
    allowance = IERC20(debtToken).allowance(msg.sender, address(this));
    if (allowance < totalDebtAmt) {
       revert TokenAllowanceInsufficient(allowance, totalDebtAmt);
    }
 } else {
     uint256 allowance = IERC20(collateralToken).allowance(msg.sender, address(this));
     if (allowance < totalCollateralAmt) {
        revert TokenAllowanceInsufficient(allowance, totalCollateralAmt);
     }
 }


 settlements[settlementId] = _settleInfo;

 emit SettlementCreated(msg.sender, settlementId);
}
```

Impact Details

Some checks are better to be added in createSettlement function.

Recommend

some checks as below can be added:

```
function createSettlement(.....) {
require(LoanInfo.debtData.feeAmt > 0, "zero fee");
require(LoanInfo.debtData.maturity > SettleInfo.expiry, "expiry higher than maturity");
require(SettleInfo.takerAmt > 0, "zero taker amount")
 //.......


}
```