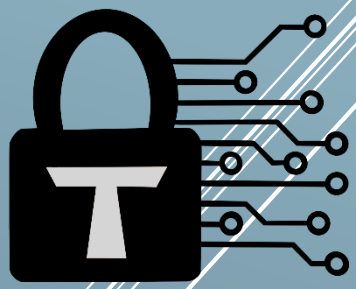
Trust Security

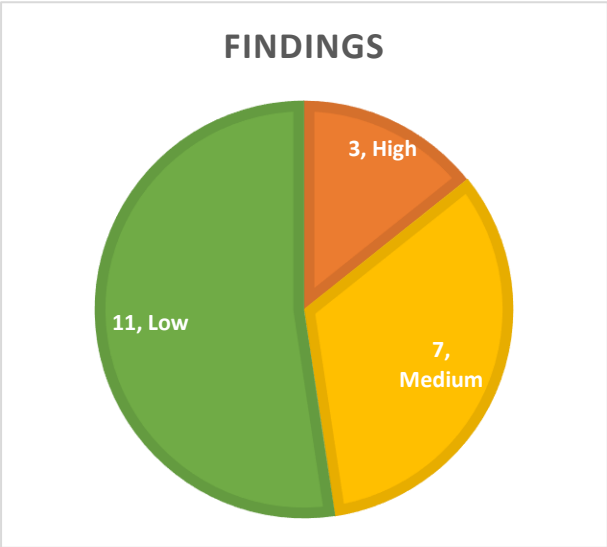


Smart Contract Audit

Coupon Finance

16/10/2023

Executive summary

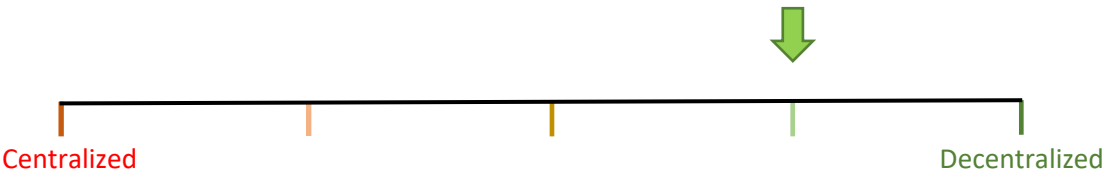


Category	Lending derivatives
Audited file count	15
Lines of Code	1569
Auditor	Trust
Time period	23/08-05/09

Findings

Severity	Total	Fixed	Acknowledged
High	3	3	-
Medium	7	7	-
Low	11	8	3

Centralization score



Signature

Digitally signed by Trust
Date: 2023-10-16 13:41:37

Document properties

Versioning

Version	Date	Description
0.1	05/09/2023	Client report
0.2	16/10/2023	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- LoanPositionManager.sol
- BorrowController.sol
- DepositController.sol
- CouponManager.sol
- OdosRepayAdapter.sol
- BondPositionManager.sol
- AaveTokenSubstitute.sol
- CouponOracle.sol
- libraries/Controller.sol
- libraries/PositionManager.sol
- libraries/Epoch.sol
- libraries/LoanPosition.sol
- libraries/BondPosition.sol
- libraries/Wrapped1155MetadataBuilder.sol
- libraries/CouponKey.sol

Repository details

- **Repository URL:** https://github.com/clober-dex/coupon-finance
- **Commit hash:** 6e917ad8f21b9b7c25469589dd733e1c1d92c87a

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to the Immunefi bug bounty platform and is currently a Code4rena judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	There are several components of considerable complexity in the protocol
Documentation	Moderate	The project code-level and high-level docs could be improved.
Best practices	Excellent	The project utilizes the latest industry standards and techniques.
Centralization risks	Good	The protocol is mostly decentralized barring the described risks.

Findings

High severity findings

TRST-H-1 All loanable assets can be stolen due to underflow in liquidate()

- **Category:** Underflow issues
- **Source:** LoanPositionManager.sol
- **Status:** Fixed

Description

The *liquidate()* function in LoanPositionManager determines the liquidation amount using *_getLiquidationAmount():*

```
unchecked {
    LoanPosition memory position = _positionMap[positionId];
    (liquidationAmount, repayAmount, protocolFeeAmount) =
        _getLiquidationAmount(position, maxRepayAmount > 0 ?
maxRepayAmount : type(uint256).max);
```

If borrower has not provided coupons for this epoch, amount is determined by this clause:

```
if (position.expiredWith.endTime() <= block.timestamp) {
    if (maxRepayAmount >= position.debtAmount) {
        repayAmount = position.debtAmount;
    } else if (maxRepayAmount + minDebtAmount > position.debtAmount)
{
        if (position.debtAmount < minDebtAmount) revert
TooSmallDebt();
        repayAmount = position.debtAmount - minDebtAmount;
    } else {
        repayAmount = maxRepayAmount;
    }
    liquidationAmount = Math.ceilDiv(
        repayAmount * debtPriceWithPrecisionComplement *
_RATE_PRECISION,
        collateralPriceWithPrecisionComplement * (_RATE_PRECISION -
loanConfig.liquidationFee)
    );
```

It aims to repay the entire debt amount if user is able to supply it. Then **liquidationAmount** is calculated from the **repayAmount** by converting the value using oracles, and applying the liquidation fee.

A crucial check is missing which is validated when the position is unhealthy, but not expired:

```
if (liquidationAmount > position.collateralAmount) liquidationAmount
= position.collateralAmount;
```

Since this check is missing, it may liquidate more than **collateralAmount**.

Back in *liquidate()*, this amount is decreased from the user's collateral without additional checks.

```
position.collateralAmount -= liquidationAmount;
position.debtAmount -= repayAmount;
if (position.debtAmount == 0) {
    position.expiredWith = currentEpoch.sub(1);
    _positionMap[positionId].expiredWith = position.expiredWith;
}
_positionMap[positionId].collateralAmount =
position.collateralAmount;
_positionMap[positionId].debtAmount = position.debtAmount;
```

The operations are done in an **unchecked{}** block. This means **collateralAmount** will underflow and be close to MAX_UINT256. With this amount, the user can easily borrow the entire Coupon asset reserve and stay at a healthy collateral ratio. This opens a self-liquidation vector, which might be mitigated by liquidator bots liquidating the position before the LTV rises so that the liquidated amount surpasses the collateral amount. However as will be shown in H-2 and H-3, an attacker can deny liquidations, guaranteeing the success of the exploit.

Recommended mitigation

Move the **liquidationAmount** check so that it will apply for expired borrowing as well.

Team response

Fixed.

Mitigation review

Mitigation has been applied correctly.

TRST-H-2 A user can abuse reentrancy to avoid liquidations leading to insolvency of the protocol

- **Category:** Reentrancy attacks
- **Source:** PositionManager.sol
- **Status:** Fixed

Description

The PositionManager implements all protocol invariants and should make inheriting contracts safe as long as they implement the *adjustPosition()* and *settlePosition()* functions correctly. Interactions with the contract begin by calling *lock()* which adds the sender to the locker array and calls their callback.

```
function lock(bytes calldata data) external returns (bytes memory
result) {
    _lockData.push(msg.sender);
    result = IPositionLocker(msg.sender).positionLockAcquired(data);
    if (_lockData.length == 1) {
```

```
        if (_lockData.nonzeroDeltaCount != 0) revert NotSettled();
        delete _lockData;
    } else {
        _lockData.pop();
    }
}
```

The key defense for position changes is that **nonzeroDeltaCount** must equal 0 when the first locker (in a possible re-entrant flow) returns from the callback. This would imply that any position that was unsettled has been re-settled, and any token imbalance has been re-balanced.

Note a key detail – the delta count is only checked on the first locker. Re-entering lockers can leave token imbalances and unlock, which would revert the first unlocker unless they cover their imbalances. While this is an obvious DOS vector if an untrusted user gets execution, it is not clear how it can lead to material damage.

However, the combination of the *lock()* properties with the *liquidate()* implementation does lead to a critical issue.

When a user is liquidated and they have borrow rights (they supplied coupons) for future epochs, these must be returned to them. It is done below:

```
if (epochLength > 0) {
    address couponOwner = ownerOf(positionId);
    Coupon[] memory coupons = new Coupon[](epochLength);
    for (uint256 i = 0; i < epochLength; ++i) {
        coupons[i] = CouponLibrary.from(position.debtToken,
currentEpoch.add(uint8(i)), repayAmount);
    }
    try ICouponManager(_couponManager).mintBatch(couponOwner,
coupons, "") {}
    catch {
        for (uint256 i = 0; i < epochLength; ++i) {
            _couponOwed[couponOwner][coupons[i].id()] +=
coupons[i].amount;
        }
    }
}
```

The code will create a coupon array and request that the CouponManager mint them for the liquidated user through *mintBatch()*.

```
function mintBatch(address to, Coupon[] calldata coupons, bytes
memory data) external onlyMinter {
    (uint256[] memory ids, uint256[] memory amounts) =
_splitCoupons(coupons);
    _mintBatch(to, ids, amounts, data);
}
```

It will call the ERC1155 *_mintBatch(),* which as the specification requires, checks a smart contract is capable of receiving the token.

```
  * - If `to` refers to a smart contract, it must implement
{IERC1155Receiver-onERC1155BatchReceived} and re
  * acceptance magic value.
  */
```

```
 function _mintBatch(address to, uint256[] memory ids, uint256[]
memory values, bytes memory data) internal
     if (to == address(0)) {
         revert ERC1155InvalidReceiver(address(0));
     }
     _updateWithAcceptanceCheck(address(0), to, ids, values, data);
 }
```

Therefore, the liquidated user (smart contract) receives arbitrary execution through the *onERC1155BatchReceived()* call. At this point, they can simply create a token imbalance in the LoanPositionManager and exit (claiming they support receiving the coupons). As explained, the delta check will be applied when the *liquidator lock()* callback finishes, which will cause a revert. Therefore, a user can dodge liquidations, which may lead to bad debt and insolvency of the protocol.

Recommended mitigation

Consider checking the **nonzeroDeltaCount** at every unlock.

Team response

Fixed.

Mitigation review

The mitigation ensures attacker cannot reenter by only invoking a callback if the user is an EOA. Note that smart contract clients must support the *claimOwedCoupons()* method to guarantee these will remain accessible.

TRST-H-3 A user can abuse gas flaws to avoid liquidations leading to insolvency of the protocol

- **Category:** Gas-related flaws
- **Source:** LoanPositionManager.sol
- **Status:** Fixed

Description

This finding relates to the *liquidate()* coupon dispatch flow described in H-2. The following code block attempts to send the liquidator their coupons.

```
try ICouponManager(_couponManager).mintBatch(couponOwner, coupons,
"") {}
catch {
    for (uint256 i = 0; i < epochLength; ++i) {
        _couponOwed[couponOwner][coupons[i].id()] +=
coupons[i].amount;
    }
}
```

If the minting fails, it saves the coupons in storage so that the liquidated user can claim them. As described, a liquidated smart contract gains arbitrary execution. In fact, they can deplete

the entire gas bank sent to them using an infinite loop, massive memory expansion or any other gas-wasting construct. According to the 1/64 rule, the user smart contract would waste 63/64 of the remaining gas. This means that for the *liquidate()* call to not revert, 1/64 of the total gas at the moment of call must suffice. However, there are expensive SSTORES needed to store **_couponOwed** array. Each will cost 22100. This means every epoch costs a minimum of 1414400 gas, plus many other operations beside the **_couponOwed** storage, making liquidation impractical.

Recommended mitigation

Pass a limited amount of gas to *mintBatch()* so that the callback would not be able to DOS the liquidation.

Team response

Fixed.

Mitigation review

The fix applied to H-2 resolves this issue.

Medium severity findings

TRST-M-1 Unsafe integration with Chainlink price feeds

- **Category:** Integration flaws
- **Source:** CouponOracle.sol
- **Status:** Fixed

Description

The CouponOracle receives prices via Chainlink price feeds as shown below

```
function getAssetPrice(address asset) public view returns (uint256) {
    address feed = getFeed[asset];
    if (feed != address(0)) {
        (, int256 price,,,) =
AggregatorV3Interface(feed).latestRoundData();
        if (price > 0) return uint256(price);
    }
    return IFallbackOracle(fallbackOracle).getAssetPrice(asset);
}
```

When interfacing with Chainlink aggregators, there are several important safety checks to incorporate:

1. Sanity checks
2. Staleness checks
3. Sequencer uptime checks

These are all well summarized in the blog post here. Apart from the **price** sanity check, all other checks are missing.

This could lead to various risks like using an invalid price feed or liquidating a user unfairly.

Recommended mitigation

Add the checks above.

Team response

Fixed.

Mitigation review

Appropriate checks have been added.

TRST-M-2 Controller functions will not operate when market is paused due to wrong burnableAmount()

- **Category:** Integration flaws
- **Source:** AaveTokenSubstitute.sol
- **Status:** Fixed

Description

The AaveTokenSubstitute contract holds aTokens and wraps them as another token. It exposes the following function, which is the amount of aTokens that can be cashed on AAVE to get underlying.

```
function burnableAmount() external view returns (uint256) {
    return IERC20(underlyingToken).balanceOf(address(aToken));
}
```

The issue is that it does not take into account that the market can be paused. In that situation, it would return a larger number than is burnable. It would lead *_burnAllSubstitute()* to attempt to burn more than possible and revert, instead of burning and redeeming aTokens. The function is called by both DepositController and BorrowController during cleanup, so this affects all user code flows.

Recommended mitigation

If the market is paused, return zero in *burnableAmount()*.

Team response

Fixed.

Mitigation review

Issue has been resolved.

TRST-M-3 Unsafe use of permit() makes user interactions vulnerable to DOS attacks

- **Category:** Frontrunning attacks
- **Source:** Controller.sol
- **Status:** Fixed

Description

The *permit()* ERC20/ERC721 extension (EIP2612, EIP4494) is widely used to save users the hassle of sending an *approve()* TX before a contract interaction. The controller uses them.

```
function _permitERC20(address token, uint256 amount, PermitParams
calldata p) internal {
    if (p.deadline > 0) {
        IERC20Permit(ISubstitute(token).underlyingToken()).permit(
            msg.sender, address(this), amount, p.deadline, p.v, p.r,
p.s
        );
    }
}
function _permitERC721(IERC721Permit permitNFT, uint256 positionId,
PermitParams calldata p) internal {
    if (p.deadline > 0) permitNFT.permit(address(this), positionId,
p.deadline, p.v, p.r, p.s);
}
```

The BorrowController and DepositController user interactions support them by passing PermitParams, for example:

```
function deposit(
    address asset,
    uint256 amount,
    uint8 lockEpochs,
    uint256 minEarnInterest,
    PermitParams calldata tokenPermitParams
) external payable nonReentrant wrapETH {
    _permitERC20(asset, amount, tokenPermitParams);
```

The issue with the *permit()* call implementation is that anyone observing the mempool is able to DOS the controller transaction. They would copy the permit params and call *permit()* directly on the ERC20/ERC721 contract. This would advance the permit nonce and when *permit()* will try authenticating the signature, it would revert.

The DOS is not permanent because a user could pass an empty **PermitParams** after it was called by an attacker, in this case the call to *permit()* is skipped.

Recommended mitigation

Wrap the *permit()* external call with a try/catch. In case of failure, **if** the user allowance is sufficient, do not revert.

Team response

Fixed.

Mitigation review

The new code will never revert due to *permit()* failure.

TRST-M-4 burnExpiredCoupons() will not burn coupons correctly

- **Category:** Logical flaws
- **Source:** CouponManager.sol
- **Status:** Fixed

Description

The CouponManager allows users to burn their expired coupons using the function below:

```
function burnExpiredCoupons(CouponKey[] calldata couponKeys) external
{
    uint256[] memory ids = new uint256[](couponKeys.length);
    uint256[] memory amounts = new uint256[](couponKeys.length);
    Epoch current = EpochLibrary.current();
    uint256 count;
    for (uint256 i = 0; i < couponKeys.length; ++i) {
        if (couponKeys[i].epoch >= current) continue;
        uint256 id = couponKeys[i].toId();
        uint256 amount = balanceOf(msg.sender, id);
        if (amount == 0) continue;
        count++;
        ids[i] = id;
        amounts[i] = amount;
    }
    assembly {
        mstore(ids, count)
        mstore(amounts, count)
    }
    _burnBatch(msg.sender, ids, amounts);
}
```

The loop will iterative over the expired coupons. If a coupon is not expired or has zero amount, it is skipped. Otherwise, the **id** and **amount** will be stored in the **i-th** entry of the corresponding array. This is an error, as **i** will keep increasing which will make the arrays have zero-initialized values for skipped coupons. The assembly level setting of the array size will trim out the actual expired tokens if they are at the end of the array.

Recommended mitigation

Store in **ids[count], amounts[count]** in the loop. Count represents the amount of valid entries.

Team response

Fixed.

Mitigation review

Suggested mitigation was applied.

TRST-M-5 Certain legitimate coupon trades will revert in controllers that use the base Controller trading function

- **Category:** Logical flaws
- **Source:** Controller.sol
- **Status:** Fixed

Description

The Controller has the *_executeCouponTrade()* function to trade coupons through the Clober DEX with certain constraints. It is structured as a recursive implementation, where the last coupon is peeled off the array, traded and then through the Clober callback the rest of the array is traded using *_executeCouponTrade()* again.

```
if (couponsToBuy.length > 0) {
    Coupon memory lastCoupon = couponsToBuy[couponsToBuy.length - 1];
    assembly {
        mstore(couponsToBuy, sub(mload(couponsToBuy), 1))
    }
    bytes memory data = abi.encode(
        user, lastCoupon, couponsToBuy, new Coupon[](0), amountToPay,
maxPayInterest, leftRequiredInterest
    );
    assembly {
        mstore(couponsToBuy, add(mload(couponsToBuy), 1))
    }
    CloberOrderBook market =
CloberOrderBook(_couponMarkets[lastCoupon.id()]);
    uint256 dy = lastCoupon.amount -
IERC20(market.baseToken()).balanceOf(address(this));
    market.marketOrder(address(this), type(uint16).max,
type(uint64).max, dy, 1, data);
```

The function receives **couponsToBuy** and **couponsToSell** arrays. When one of them is not empty, it assumes the other one is (an empty **Coupon[]** array is passed instead of the actual array).

This assumption holds when users interact with the BorrowController or DepositController. However as Controller was written with inheritance in mind, it needs to handle the generic case when coupons need to be both bought and sold. The buy and sell lists come from the Position *calculateCouponRequirement()* function. This is the critical code component which determines the coupon array lengths.

```
uint256 mintCouponsLength =
newPosition.expiredWith.sub(latestExpiredEpoch);
uint256 burnCouponsLength =
oldPosition.expiredWith.sub(latestExpiredEpoch);
unchecked {
    uint256 minCount = Math.min(mintCouponsLength,
burnCouponsLength);
    if (newPosition.amount > oldPosition.amount) {
        burnCouponsLength -= minCount;
    } else if (newPosition.amount < oldPosition.amount) {
        mintCouponsLength -= minCount;
    } else {
        mintCouponsLength -= minCount;
        burnCouponsLength -= minCount;
```

```
    }
}
```

Consider the possibility that a user wishes to extend the debt duration and lower the debt amount. In this case, for the epochs up to the previous expiry epoch, a user is eligible for coupons (mintCoupons), after which they will need to burn coupons until the new expiry epoch. Both arrays will not be empty.

The impact will be that the described position changes cannot be implemented in controllers (will revert), although they use the Controller parent code correctly.

Recommended mitigation

Pass the original arrays to the recursive implementation instead of zero-sized arrays in *_executeCouponTrade()*.

Team response

Fixed.

Mitigation review

Suggested mitigation has been applied.

TRST-M-6 Repays through the Odos adapter could fail

- **Category:** Logical flaws
- **Source:** OdosRepayAdapter.sol
- **Status:** Fixed

Description

In the OdosRepayAdapter, a user can reduce their collateral to pay off debt. It will first remove collateral, swap it using an Odos call, adjust the position and then sell the minted coupons and pay off more debt.

It will prevent depositing too many debt tokens using this check:

```
if (position.debtAmount < depositDebtTokenAmount) {
    depositDebtTokenAmount = position.debtAmount;
}
```

However, it is missing a check could trip the execution in *settlePosition()*:

```
if (position.debtAmount > 0 && minDebtAmount > position.debtAmount)
revert TooSmallDebt();
```

In BorrowController it is not necessary, since the user knows ahead of time how much debt will be repaid. However, In the OdosRepayAdapter, this depends on coupon and collateral swaps.

Recommended mitigation

In case the remaining debt would be smaller than **minDebtAmount**, reduce the repay amount accordingly.

Team response

Fixed.

Mitigation review

Issue has been resolved correctly.

TRST-M-7 An attacker may be able to make the OdosRepayAdapter unusable

- **Category:** ERC20 compatibility issues
- **Source:** OdosRepayAdapter.sol
- **Status:** Fixed

Description

There are several popular tokens (e.g. USDT) which require *approve()* calls to alternative between zero and non-zero (this is to mitigate a well known double-spending issue).

In *_swapCollateral()* in OdosRepayAdapter, it approves the input amount for the Odos router and calls it with the provided calldata. The calldata is meant to transfer into the router the input amount.

```
IERC20(inToken).approve(_odosRouter, inAmount);
(bool success, bytes memory result) = _odosRouter.call(swapData);
```

The issue is that a malicious user can pass input data that transfers into the router a smaller amount than **inAmount**. In this case, there will remain a non-zero approval for the router. In the next invocation of *_swapCollateral()*, the approve shall revert.

Recommended mitigation

Either use OpenZeppelin's *safeApprove()* method, or call *approve(_odosRouter,0)* after the router call.

Team response

Fixed.

Mitigation review

Suggestion mitigation has been implemented.

Low severity findings

TRST-L-1 The Epoch contract pragma statement is incorrect

- **Category:** Compilation issues
- **Source:** Epoch.sol
- **Status:** Fixed

Description

The Epoch contract declares it requires compilation with at least an 0.8.0 compiler.

```
pragma solidity ^0.8.0;

type Epoch is uint8;

using {gt as >, gte as >=, lt as <, lte as <=, eq as ==} for Epoch
global;
```

However, it uses user-defined operators which require **0.8.19**.

Recommended mitigation

Change the pragma statement according to the Solidity version required.

Team response

Fixed.

Mitigation review

Fixed.

TRST-L-2 Epoch functions are misleading

- **Category:** Time-sensitivity issues
- **Source:** Epoch.sol
- **Status:** Fixed

Description

The Epoch contract implements two complementary functions – *timestampToEpoch()* and *epochToTimestamp()*. These are used to get the current epoch as well as the end time of the current epoch. In Coupon, an epoch is an identifier for a 6 month period, beginning on Jan 1, 1970.

It has been observed that the two functions will not be consistent on the **last second** of an epoch. For an **x** which is the last second, **timetampToEpoch(epochToTimestamp(x)) = x+1**. Although a practical exploit has not been found, there could be severe risks involved.

Recommended mitigation

The last second of an epoch should not count as the next epoch.

Team response

Fixed.

Mitigation review

Suggestion fix has been implemented.

TRST-L-3 ETH should only be accepted when WETH is the handled asset

- **Category:** Validation issues
- **Source:** DepositController.sol, BorrowController.sol
- **Status:** Acknowledged

Description

The controller functions use the *wrapETH()* modifier so that native ETH can be accepted when debt or deposit is WETH. When it is used in such cases, any spare ETH would be returned to the user in this line:

```
_burnAllSubstitute(asset, msg.sender);
```

However, if the user accidentally sends native ETH when the handled asset is *not* WETH, the contract would consume the tokens and user would lose them permanently. They will be waiting for the next user interaction with WETH, at which point they will be gifted to them.

Recommended mitigation

Validate that the handled asset is WETH whenever the *wrapETH()* modifier is used.

Team response

Acknowledged.

TRST-L-4 Possible overflow in LoanPositionManager would zero out minDebtAmount and allow leaving small debt

- **Category:** Validation issues
- **Source:** LoanPositionManager.sol
- **Status:** Fixed

Description

The LoanPositionManager calculates the current collateral and debt token prices, and will normalize their decimals in *_calculatePricesAndMinDebtAmount()*

We can see the decimal normalization for **minDebtAmount** here:

```
uint256[] memory prices =
ICouponOracle(oracle).getAssetsPrices(assets);
```

```
// @dev `decimal` is always less than or equal to 18
minDebtAmount = (minDebtValueInEth * prices[2]) / 10 ** (18 -
debtDecimal) / prices[1];
```

The function assumes **debtDecimal <= 18**, however this is **not** enforced at the LoanConfiguration setup. If it is not true, due to the **unchecked{}** statement there will be an overflow and **minDebtAmount** would be zero-ed out.

Recommended mitigation

Consider verifying any assumptions around decimals in the *setLoanConfiguration()* function.

Team response

Fixed.

Mitigation review

Issue has been addressed.

TRST-L-5 The expiredWith parameter could be ignored in PositionManagers

- **Category:** Validation issues
- **Source:** LoanPositionManager.sol, BondPositionManager.sol
- **Status:** Acknowledged

Description

The *adjustPosition()* functions in PositionManagers receive a new position amount and expiry. In case the new amount is zero, the expiry is automatically filled to be the last expiry epoch.

```
_positionMap[positionId].expiredWith = debtAmount == 0 ?
lastExpiredEpoch : expiredWith;
```

This could surprise users which pass a different value for **expiredWith**. There is no reason to ignore user's parameter.

Recommended mitigation

Validate that the passed parameter is the **lastExpiredEpoch**, instead of overriding it which could lead to user errors.

Team response

Acknowledged.

TRST-L-6 Anyone can DOS the controller if the AAVE market is frozen

- **Category:** DOS attacks
- **Source:** Controller.sol

- **Status:** Fixed

Description

The different Controller user functions call *_ensureBalance()* through the coupon trade function. If there is an underlying token balance (USDC, WETH…), it shall be converted to Substitute tokens through *mint()*.

```
if (underlyingBalance > 0) {
    IERC20(underlyingToken).approve(token, underlyingBalance);
    ISubstitute(token).mint(underlyingBalance, address(this));
}
```

mint() will supply the asset to the market.

```
function mint(uint256 amount, address to) external {
    IERC20(underlyingToken).safeTransferFrom(msg.sender,
address(this), amount);
    IERC20(underlyingToken).approve(address(_aaveV3Pool), amount);
    _aaveV3Pool.supply(underlyingToken, amount, address(this), 0);
    _mint(to, amount);
}
```

Note that AAVE markets disallow *supply()* but allow *withdraw()* when frozen (in active/paused state they behave identically). Therefore, in a scenario where the market is frozen, but user wishes to withdraw through the controller, an adversary can donate a tiny amount of underlying to trigger the *supply()* flow. At this point, the entire TX will revert.

This makes the Coupon market have a hidden pause function on top of the AAVE dependency.

Since likelihood is low and impact is not long-term (A controller can be replaced with no risk of funds), this is regarded as Low-severity.

Recommended mitigation

If the current balance satisfies **amount**, do not revert when unable to mint Substitute tokens.

Team response

Fixed.

Mitigation review

The code has been refactored to address various concerns.

TRST-L-7 Substitute token returns wrong mintable amount

- **Category:** Logical flaws
- **Source:** AaveTokenSubstitute.sol
- **Status:** Fixed

Description

The Substitute token has a function which returns the mintable aToken amount, taking into consideration the supply cap of the market.

```
function mintableAmount() external view returns (uint256) {
    DataTypes.ReserveConfigurationMap memory configuration =
        _aaveV3Pool.getReserveData(underlyingToken).configuration;
    return configuration.getSupplyCap() * 10 **
(configuration.getDecimals());
}
```

The issue is that the function does not account for already existing tokens.

Recommended mitigation

Decrease the supply cap by the balance of aTokens.

Team response

Fixed.

Mitigation review

The code has been refactored to address various concerns.

TRST-L-8 When an empty position is liquidated, it will not be burnt

- **Category:** Logical flaws
- **Source:** LoanPositionManager.sol
- **Status:** Fixed

Description

When a position is settled in LoanPositionManager, and the **debtAmount** and **collateralAmount** is zero, it is burnt.

```
if (position.debtAmount == 0 && position.collateralAmount == 0)
_burn(positionId);
```

However, this check is missing in the *liquidate()* flow. Since liquidations do not need a subsequent *settlePosition()*, they will never be burnt.

Recommended mitigation

Add a similar check in the *liquidate()* function.

Team response

Fixed.

Mitigation review

Suggestion mitigation has been implemented.

TRST-L-9 The BorrowController and Odos Adapter do not approve debt tokens properly

- **Category:** Logical flaws

- **Source:** OdosRepayAdapter.sol
- **Status:** Fixed

Description

In the OdosRepayAdapter, it deposits debtTokens into the LoanPositionManager to reduce debt:

```
_loanManager.depositToken(position.debtToken,
depositDebtTokenAmount);
```

However, there is no *approve()* call to allow the manager to transfer those tokens. It relies on the *setCollateralAllowance()*, which is supposedly only supposed to be used to allow collateral.

```
function setCollateralAllowance(address collateralToken) external
onlyOwner {
    IERC20(collateralToken).approve(address(_loanManager),
type(uint256).max);
}
```

For the test suite, the collateral and debt tokens fully interlap, making the transfers safe, however it will not hold for all possible configurations.

Recommended mitigation

Rename the *setCollateralAllowance()* function so that it can be intentionally used for both collateral and debt tokens.

Team response

Fixed.

Mitigation review

Issue has been addressed.

TRST-L-10 Users that have an expired position can never recover

- **Category:** Logical flaws
- **Source:** LoanPositionManager.sol
- **Status:** Fixed

Description

The *settlePosition()* function does not allow a position owner to repay their debt or increase the duration if the expiry date has passed.

```
if (position.debtAmount > 0 && position.expiredWith <=
EpochLibrary.lastExpiredEpoch()) revert UnpaidDebt();
```

This seems to be an overly strict approach.

Recommended mitigation

Consider allowing a user to come back into a healthy, unliquidatable position.

Team response

Fixed.

Mitigation review

New implementation supports position recovery.

TRST-L-11 Scammers may fake Coupons and sell them for profit

- **Category:** Logical flaws
- **Source:** Wrapped1155MetadataBuilder.sol
- **Status:** Acknowledged

Description

The Coupon architecture relies on the wrapping of Coupons, which are ERC1155 tokens, into ERC20 tokens, for DEX trading. The ERC20 properties of symbol, name and decimals are controlled by the depositor of the coupon into the ERC1155 wrapper. This is done by the *buildWrapped1155Metadata()* function when operating with the controllers, but can easily be hand-built by a user.

This opens the opportunity for a scam market, where tokens wrapping one type of coupon could appear to represent another, higher-valued coupon. It is very hard to distinguish between the two, and requires querying the Wrapped1155Factory contract's *getWrapped1155()* and comparing the result with the suspected scam token.

Recommended mitigation

Consider customizing the Wrapped1155Factory and add safeguards to make scam tokens harder to craft.

Team response

Acknowledged.

Additional recommendations

TRST-R-1 Emit events on important state updates

In changes of important state variables, especially in maintenance functions, it is important to emit an event so that the contract can be easily tracked via indexers.

Consider adding events at least in the following functions:

- *setFallbackOracle()*
- *setCollateralAllowance()*
- *setTreasury()*
- *setCouponMarket()*

TRST-R-2 Guard from potential overflow

The following function in LoanPositionManager can overflow:

```
unchecked {
    for (uint256 i = 0; i < couponsToMint.length; ++i) {
        _accountDelta(couponsToMint[i].id(), 0,
couponsToMint[i].amount);
    }
    for (uint256 i = 0; i < couponsToBurn.length; ++i) {
        _accountDelta(couponsToBurn[i].id(), couponsToBurn[i].amount,
0);
    }
    collateralDelta = _accountDelta(
        uint256(uint160(oldPosition.collateralToken)),
collateralAmount, oldPosition.collateralAmount
    );
    debtDelta = -
_accountDelta(uint256(uint160(oldPosition.debtToken)),
oldPosition.debtAmount, debtAmount);
}
```

debtDelta and the return value of *_accountDelta()* are int256. When flipping the sign of the lowest possible int256 value, an overflow is triggered which would making the flip a no-op.

In this scenario, no real harm is done as the **debtDelta** is consumed by a controller or user contract, and should be well beyond any practical use.

TRST-R-3 Accept a minimum liquidation amount

In *liquidate()*, a liquidator passes the **maxRepay**. It would make sense to also support a **minLiquidationAmount**. The contract can quickly revert instead of spending additional gas in case the liquidation amount is insufficient.

TRST-R-4 Support withdrawal epoch changes in DepositController

It would make sense for users to be interested in shortening the duration of a deposit.

TRST-R-5 Consider the use of BUSL licensed code

The PositionManager contract holds some striking similarities to a core Uniswap-v4 contract. It is important to consider the use of this BUSL-licensed code from all angles.

Centralization risks

TRST-CR-1 All loanable assets can be stolen if the owner key is compromised

The privileged *setLoanConfiguration()* function in the LoanPositionManager can introduce new (collateral,debt) combinations without restrictions. A malicious admin could abuse it to allow borrowing against worthless tokens and drain the AssetPool.

TRST-CR-2 Fallback oracle can be changed

The admin is able to change the fallback oracle. It can only be used when the main oracle is invalid. However, being able to instantly change it introduces centralization risk when the primary oracle is discontinued, for any reason.

Systemic risks

TRST-SR-1 Coupon relies on AAVE liquidity

Assets deposited into Coupon Finance are wrapped to gain yield on AAVE v3. If AAVE markets are frozen, go into bad debt, or lack sufficient reserves, it will directly impact users of Coupon.

TRST-SR-2 AAVE yield all goes to the treasury

Users should be aware of the fact the yield from deposited collateral all goes to the Coupon treasury. The coupon buyers (borrowers) must leave an amount of non-yielding collateral, which will be reflected in the amount they would be willing to bid on coupons. Coupon price should show the market's view of the fixed lending rate, minus the opportunity cost of the collateral value yield over that epoch.