Sec3™

Security Assessment Report

TeaREX

February 22, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the TeaREX smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 24 issues or questions.

program	type	commit
Tea-REX	Solidity	5bcac9684c427b42392dc5a75223a8bd734e1d55

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
TEA-REX		
[H-01] Incorrect implementation prevents interest collection	High	Resolved
[H-02] Inconsistent protocol fee decimals	High	Resolved
[H-03] The repay function incorrectly calculates interest losses	High	Resolved
[H-04] Position "swappableAmount" not updated	High	Resolved
[H-05] "suppliedConversionRate" not updated during force close	High	Resolved
[H-06] Unchecked "_market" in the "TradingCore" contract	High	Resolved
[M-01] Improper asset value decimal normalization	Medium	Resolved
[M-02] "_openPosition" not transfer fees to "_feeConfig.treasury"	Medium	Resolved
[M-03] Incorrect price check in "ClosePosition" function	Medium	Acknowledged
[M-04] "onlyNotPaused" check not implemented in "TradingCore"	Medium	Resolved
[M-05] Incorrect TWAP price formula	Medium	Resolved
[L-01] Unsafe cast in "ChainlinkOracle"	Low	Resolved
[L-02] Incorrect rounding direction in function "commitBorrow"	Low	Resolved
[L-03] Interest fee not collected when updating reserve ratio	Low	Resolved
[L-04] Protocol fee not collected during "FeeConfig" update	Low	Acknowledged
[L-05] Inaccurate liquidatable check in "_ensureNotLiquidated"	Low	Resolved
[L-06] Potential overflow in TWAP oracle	Low	Acknowledged
[L-07] Price update time not checked	Low	Resolved
[L-08] Check "leverage" in "closePosition"	Low	Resolved
[I-01] Code style optimization	Info	Resolved
[I-02] The "repay" function does not check if the "debtInfo" is already closed	Info	Resolved
[I-03] Validate parameters in "setSupplyCap" and "setBorrowCap"	Info	Resolved
[I-04] Check loss ratio in function "addMargin"	Info	Acknowledged
[I-05] Avoid using user-provided "calldataProcessor" address	Info	Acknowledged

Findings in Detail

TEA-REX

[H-01] Incorrect implementation prevents interest collection

In the “Pool” contract, the “_calculateSuppliedRate” function is used to calculate the new supplied conversion rate. The result “newSuppliedConversionRate” determines how many underlying assets one TeaToken can be exchanged for.

```
/* contracts/lending/Pool.sol */
392 | function _calculateSuppliedRate(bool _isIncrease, uint256 _amountDelta) internal view returns
 ↪  (uint256 rate) {
393 |     rate = suppliedConversionRate;
395 |     if (_amountDelta > 0) {
396 |         uint256 rateDelta = _amountDelta / totalSupply();
397 |         _isIncrease ? rate + rateDelta : rate - rateDelta;
398 |     }
399 | }
```

However, the calculation result on line 397 is never assigned to the “rate” variable, meaning the computed value is never incorporated into “rateDelta”.

This omission results in an inaccurate “newSuppliedConversionRate”, causing TeahouseFinance’s economic model to malfunction.

Scenario

1. The victim supplies 1 USD to the pool.
2. After waiting for a period of time, the victim attempts to withdraw and expects to receive interest income.

Due to an error in the “_calculateSuppliedRate” method, the “newSuppliedConversionRate” calculation in the “withdraw” function fails to properly accumulate interest. This issue occurs through the following sequence of method calls: “withdraw” → “_collectInterestFeeAndCommit” → “_collectInterestAndFee” → “_calculateSuppliedRate”.

```
/* contracts/lending/Pool.sol */
189 | function withdraw(
190 |     address _account,
191 |     address _withdrawTo,
192 |     uint256 _amount
193 | ) external override nonReentrant onlyNotPaused onlyRouter returns (
199 |     (, , uint256 newSuppliedConversionRate, ) = _collectInterestFeeAndCommit();
204 |     withdrawnUnderlying = _toUnderlying(_amount, newSuppliedConversionRate, false);
219 |     _underlyingAsset.safeTransfer(_withdrawTo, withdrawnUnderlying - withdrawalFee);
```

The victim received the principal but did not receive any interest income.

Recommendations

Consider changing line 397 from:

```
/* contracts/lending/Pool.sol */
397 |     _isIncrease ? rate + rateDelta : rate - rateDelta;
```

to

```
_isIncrease ? rate += rateDelta : rate -= rateDelta;
```

Resolution

Fixed by commit "`6689af3`".

TEA-REX

[H-02] Inconsistent protocol fee decimals

According to the protocol design, when a user borrows the underlying asset, the corresponding updated "`borrowedTeaToken`" amount will be calculated as

"`underlyToken.decimal * DECIMALS_MULTIPLIER (1e18)`"

For example, if a user borrows "`1e6`" USDC, the "`borrowedTeaToken`" amount will increase by "`1e24`".

Additionally, the state variable "`pendingFee`" will record a portion of the "`borrowedTeaToken`" as the protocol fee, so the fee's decimal format will also follow

"`underlyToken.decimal * DECIMALS_MULTIPLIER`"

```
/* contracts/lending/Pool.sol */
function _collectInterestAndFee() internal view returns (
    uint256 interest,
    uint256 fee,
    uint256 newSuppliedConversionRate,
    uint256 newBorrowedConversionRate
) {
    uint256 timeElapsed = block.timestamp - lastAccumulateTimestamp;
    uint256 _borrowedTeaToken = borrowedTeaToken;
    if (_borrowedTeaToken == 0) return (interest, fee, suppliedConversionRate,
↪ borrowedConversionRate);
    uint256 suppliedTeaToken = totalSupply();
    uint256 suppliedUnderlying = suppliedTeaToken.mulDiv(suppliedConversionRate, RATE_MULTIPLIER);
    uint256 borrowedUnderlying = _borrowedTeaToken.mulDiv(borrowedConversionRate, RATE_MULTIPLIER);
    uint256 _borrowedConversionRate = borrowedConversionRate;
    uint256 interestRate =
↪ IInterestRateModel(router.getInterestRateModel(interestRateModelType)).getBorrowRate(
        suppliedUnderlying,
        borrowedUnderlying,
        reserveRatio
    );
    uint256 rateDeltaInterest = _calculateInterests(_borrowedConversionRate, interestRate,
↪ timeElapsed);
    uint256 rateDeltaFee = _calculateInterests(_borrowedConversionRate,
↪ router.getFeeConfig().borrowFee, timeElapsed);
    interest = _borrowedTeaToken.mulDiv(rateDeltaInterest, RATE_MULTIPLIER);
    fee = _borrowedTeaToken.mulDiv(rateDeltaFee, RATE_MULTIPLIER);
    newSuppliedConversionRate = _calculateSuppliedRate(true, interest);
    newBorrowedConversionRate = _borrowedConversionRate + rateDeltaInterest + rateDeltaFee;
}

/* contracts/lending/Pool.sol */
(interest, fee, newSuppliedConversionRate, newBorrowedConversionRate) = _collectInterestAndFee();
lastAccumulateTimestamp = block.timestamp;
suppliedConversionRate = newSuppliedConversionRate;
borrowedConversionRate = newBorrowedConversionRate;

if (fee > 0) {
    pendingFee = pendingFee + fee;
    emit BorrowFeeAccumulated(block.timestamp, fee);
}
```

However, the "`repay`" function and the "`claimFee`" function incorrectly interpret the "`pendingFee`" decimal as being the same as the underlying token's decimal, which creates an inconsistency. This discrepancy disrupts the proper functioning of fee-related protocol mechanisms, rendering them ineffective.

```
/* contracts/lending/Pool.sol */
else if (_forceClose) {
    _debtInfo.isClosed = true;
    uint256 loss = _toUnderlying(borrowedTeaToken, newBorrowedConversionRate, false);
    uint256 _pendingFee = pendingFee;
    if (loss > _pendingFee) {
        newSuppliedConversionRate = _calculateSuppliedRate(false, loss - _pendingFee);
        pendingFee = 0;
    }
    else {
        pendingFee = _pendingFee - loss;
    }
}

/* contracts/lending/Pool.sol */
function claimFee() external override nonReentrant onlyNotPaused returns (uint256 claimedFee, uint256 unclaimedFee) {
    _collectInterestFeeAndCommit();

    ERC20PermitUpgradeable _underlyingAsset = underlyingAsset;
    uint256 balance = _underlyingAsset.balanceOf(address(this));
    unclaimedFee = pendingFee;
    claimedFee = balance > unclaimedFee ? unclaimedFee : balance;

    if (claimedFee > 0) {
        unclaimedFee = unclaimedFee - claimedFee;
        pendingFee = unclaimedFee;
        address treasury = router.getFeeConfig().treasury;
        _underlyingAsset.safeTransfer(treasury, claimedFee);

        emit FeeClaimed(treasury, claimedFee);
    }
}
```

It's recommended to use a consistent "`pendingFee`" decimal.

Resolution

Fixed by commits "`6cf14a9`" and "`521d32b`".

To prevent small amount to be accounted as 0, "`ceil`" is used to carry when performing "`mulDiv`" in interest and fee calculations.

TEA-REX
[H-03] The repay function incorrectly calculates interest losses

When a user refuses to repay the remaining loan, the "`repay`" function of the "`Pool`" contract will calculate the impact of this unrecoverable loan on the accrued loan interest.

```
/* contracts/lending/Pool.sol */
336 | uint256 loss = _toUnderlying(borrowedTeaToken, newBorrowedConversionRate, false);
337 | uint256 _pendingFee = pendingFee;
338 | if (loss > _pendingFee) {
339 |     newSuppliedConversionRate = _calculateSuppliedRate(false, loss - _pendingFee);
340 |     pendingFee = 0;
341 | }
```

Line 336 was incorrectly written as "`borrowedTeaToken`", which represents the total amount borrowed by the entire Pool, leading to an unexpected amplification of bad debt losses.

This error ultimately resulted in incorrect calculations in "`_calculateSuppliedRate`", affecting the normal financial model of the Pool.

The calculation should instead account for the portion of the debt that the current user refuses to repay, specifically "`_debtInfo.borrowedTeaToken`".

Consider changing line 336 to:

```
uint256 loss = _toUnderlying(_debtInfo.borrowedTeaToken, newBorrowedConversionRate, false);
```

Resolution

Fixed by commit "`6a1bc54`".

TEA-REX

[H-04] Position "swappableAmount" not updated

During the position opening process, if the "margin" token equals the "asset" token, the "swappableAmount" is updated to include both "_assetAmount" and "_marginAmount".

```
/* contracts/trading/MarketNFT.sol */
positions[positionId] = Position({
    status: PositionStatus.Open,
    isLongToken0: _isLongToken0,
    isMarginAsset: isMarginAsset,
    stopLossRateTolerance: _stopLossRateTolerance,
    initialLeverage: leverage,
    marginAmount: _marginAmount,
    interestRateModelType: _interestRateModelType,
    borrowId: _borrowId,
    assetAmount: _assetAmount,
    swappableAmount: isMarginAsset ? _assetAmount + _marginAmount : _assetAmount,
    takeProfit: _takeProfit,
    stopLoss: _stopLoss
});
```

However, in the "_addMargin" function, when a user deposits additional margin tokens into the position, the "swappableAmount" is not updated. This inconsistency results in an incorrect "swappableAmount", potentially leading to the position being closed incorrectly.

```
/* contracts/trading/TradingCore.sol */
function _addMargin(
    MarketNFT _market,
    ERC20PermitUpgradeable _asset,
    ERC20PermitUpgradeable _debt,
    uint256 _positionId,
    IMarketNFT.Position memory _position,
    uint256 _addedAmount
) internal {
    if (_addedAmount == 0) revert ZeroNotAllowed();
    _market.addMargin(_positionId, _addedAmount);
    (_position.isMarginAsset ? _asset : _debt).safeTransferFrom(msg.sender, address(this),
↪  _addedAmount);
    emit AddMargin(_market, _positionId, _addedAmount);
}

/* contracts/trading/MarketNFT.sol */
function addMargin(
    uint256 _positionId,
    uint256 _addedAmount
) external override nonReentrant onlyNotPaused onlyTradingCore {
    Position storage position = positions[_positionId];
    if (position.status != PositionStatus.Open) revert InvalidPositionStatus();
    position.marginAmount = position.marginAmount + _addedAmount;
}
```

It's recommended to update the "`swappableAmount`" in the "`_addMargin`" function.

Resolution

Fixed by commit "`87e40ba`".

TEA-REX

[H-05] "suppliedConversionRate" not updated during force close

In the “`repay`” function, if the loss exceeds the pending fee amount, the protocol calculates a new “`newSuppliedConversionRate`” to allocate loss among all users in the pool.

However, the local variable “`newSuppliedConversionRate`” is not updated to the state variable “`suppliedConversionRate`”, rendering the debt allocation ineffective.

```
/* contracts/lending/Pool.sol */
else if (_forceClose) {
    _debtInfo.isClosed = true;
    uint256 loss = _toUnderlying(borrowedTeaToken, newBorrowedConversionRate, false);
    uint256 _pendingFee = pendingFee;
    if (loss > _pendingFee) {
        newSuppliedConversionRate = _calculateSuppliedRate(false, loss - _pendingFee);
        pendingFee = 0;
    }
    else {
        pendingFee = _pendingFee - loss;
    }
}

/* contracts/lending/Pool.sol */
function _calculateSuppliedRate(bool _isIncrease, uint256 _amountDelta) internal view returns
↪ (uint256 rate) {
    rate = suppliedConversionRate;
    if (_amountDelta > 0) {
        uint256 rateDelta = _amountDelta / totalSupply();
        _isIncrease ? rate + rateDelta : rate - rateDelta;
    }
}
```

it’s recommended to update the local variable “`newSuppliedConversionRate`” to the state variable “`suppliedConversionRate`”.

Resolution

Fixed by commit “`0d8a8be`”.

TEA-REX

[H-06] Unchecked "_market" in the "TradingCore" contract

The “`closePosition()`” in the “`TradingCore`” contract closes a specific position in the market.

```
/* contracts/trading/TradingCore.sol */
function closePosition(
    address _market,
    uint256 _positionId,
    uint256 _assetTokenToSwap,
    uint256 _minDecreasedDebtAmount,
    ICalldataProcessor _calldataProcessor,
    address _swapRouter,
    bytes calldata _data
) external override nonReentrant whenNotPaused returns (
```

However, it does not check if the “`_market`” parameter is a legitimate “`MarketNFT`” contract, which could lead to the following exploits:

Exploit 1

In the “`_closePosition`” function, the payment to the user is calculated by “`market.closePosition()`”. A malicious user could forge the return value and drain all assets from the “`tradingCore`” contract.

```
/* contracts/trading/TradingCore.sol */
(isFullyClosed, decreasedMarginAmount, owedAsset, owedDebt) = market.closePosition(
    _mode,
    _positionId,
    swappedAssetToken,
    decreasedDebtAmount,
    tradingFee,
    _router.debtOfUnderlying(debt, position.interestRateModelType, position.borrowId)
);
_pay(asset, address(this), positionOwner, owedAsset);
_pay(debt, address(this), positionOwner, owedDebt);
```

Exploit 2

In the “`openPosition`” function, the user position insolvency check relies on “`_market.openPosition()`”.

A malicious user could borrow all assets from the lending pool and bypass the insolvency check by using a fake market. Since it is a fake market/position, the borrowed assets will be stuck in the “tradingCore” contract.

```
/* contracts/trading/TradingCore.sol */
IRouter _router = router;
address pool = _router.borrow(debt, _interestRateModelType, _borrowAmount);
FeeConfig memory _feeConfig = _getFeeForAccount(msg.sender);
uint256 tradingFee = _calculateTradingFee(false, _borrowAmount, _feeConfig);
_collectTradingFee(debt, tradingFee, _feeConfig);
(uint256 debtAmount, uint256 assetAmount) = _swap(
    debt,
    asset,
    _borrowAmount - tradingFee,
    _minAssetAmount,
    _swapRouter,
    _data
);
debtAmount = debtAmount + tradingFee;
uint256 unusedAmount = _borrowAmount - debtAmount;
if (unusedAmount > 0) {
    debt.safeTransfer(pool, unusedAmount);
}
uint256 borrowId = _router.commitBorrow(debt, _interestRateModelType, debtAmount);
_margin.safeTransferFrom(msg.sender, address(this), _marginAmount);
positionId = _market.openPosition(
    msg.sender,
    _interestRateModelType,
    borrowId,
    isLongToken0,
    _marginAmount,
    debtAmount,
    assetAmount,
    _takeProfit,
    _stopLoss,
    _stopLossRateTolerance
);
```

Recommendations

Consider adding the following constraint in functions “_beforeModifyOpeningPosition”, “openPosition” and “openPositionPermit”.

```
require(pairMarket[token0.address][token1.address] == IMarketNFT(_market), InvalidMarketPair());
```

Resolution

Fixed by commit “f0683d2”.

TEA-REX

[M-01] Improper asset value decimal normalization

In the "`getPrice()`", assuming "`mulDecimals`" is greater than "`divDecimals`", the return value is:

$$\frac{assetPrice \times 10^{mulDecimals - divDecimals}}{basePrice}$$

This is equivalent to:

$$\frac{assetPrice \times 10^{mulDecimals}}{basePrice \times 10^{divDecimals}}$$

Simplified further, it becomes:

$$\frac{assetPrice \times 10^{baseTotalDecimal} \times 10^{priceDecimals}}{basePrice \times 10^{AssetTotalDecimals}}$$

After eliminating the "`priceDecimal`" factor from the "`TotalDecimal`", it becomes:

$$\frac{assetPrice\_WithoutDecimal \times 10^{baseTokenDecimal} \times 10^{priceDecimals}}{basePrice\_WithoutDecimal \times 10^{AssetTokenDecimal}}$$

```
/* contracts/trading/ChainlinkOracle.sol */
function getPrice(address _asset) external view returns (uint256 price) {
    OracleInfo storage assetInfo = oracleInfo[IERC20Metadata(_asset)];
    require(address(assetInfo.priceOracle) != address(0), AssetNotEnabled());
    if (_asset == address(baseAsset)) {
        return 10 ** priceDecimals;
    }
    OracleInfo storage baseInfo = oracleInfo[baseAsset];
    (,int256 assetPrice,,,) = assetInfo.priceOracle.latestRoundData();
    (,int256 basePrice,,,) = baseInfo.priceOracle.latestRoundData();
    uint256 mulDecimals = baseInfo.totalDecimals + priceDecimals;
    uint256 divDecimals = assetInfo.totalDecimals;
    if (mulDecimals > divDecimals) {
        price = Math.mulDiv(uint256(assetPrice), 10 ** (mulDecimals - divDecimals), uint256(basePrice));
    }
    else {
        price = uint256(assetPrice) / (10 ** (divDecimals - mulDecimals)) / uint256(basePrice);
    }
}

/* contracts/trading/ChainlinkOracle.sol */
function _addAsset(IERC20Metadata _asset, AggregatorV3Interface _priceOracle) internal {
    require(address(_asset) != address(0), InvalidAssetAddress());
    OracleInfo storage info = oracleInfo[_asset];
    info.priceOracle = _priceOracle;
    info.assetDecimals = _asset.decimals();  // token is assumed to have decimals() function
    info.priceDecimals = _priceOracle.decimals();
    info.totalDecimals = info.assetDecimals + info.priceDecimals;
}
```

When used in the “`getTokenValue()`”, the “`priceDecimals`” is eliminated. The result becomes:

$$tokenAmount\_AssetToken \times \frac{assetPrice\_WithoutDecimal}{basePrice\_WithoutDecimal} \times \frac{10^{baseTokenDecimal}}{10^{AssetTokenDecimal}}$$

```
/* contracts/trading/MarketNFT.sol */
function _getTokenValue(
    uint8 _oracleDecimals,
    uint256 _tokenAmount,
    uint256 _tokenPrice
) internal pure returns (
    uint256 value
) {
    value = _tokenAmount.mulDiv(_tokenPrice, 10 ** _oracleDecimals );
}
```

The “`getTokenValue`” function is used to calculate the supply or borrow asset value in the protocol. The term $tokenAmount_{AssetToken} \times \frac{assetPrice_{WithoutDecimal}}{basePrice_{WithoutDecimal}}$ represents the value of the asset token based on the baseToken price, typically denominated in USD. The factor $\frac{10^{baseTokenDecimal}}{10^{AssetTokenDecimal}}$ ensures that the final value of different decimal asset tokens is consistently based on “`baseTokenDecimal`”.

Thus, the expected asset token value is:

$$AssetTokenValue \times 10^{baseTokenDecimal}$$

Issues in the Current Implementation

1. Inconsistent base tokens:

The protocol assumes all price oracles use identical “`baseToken`” specifications. Disparities in either price feeds or decimal precision between different “`baseToken`” implementations could yield inconsistent asset valuations.

Consider the following example when the asset token is SEI and the base token is USDC.

- SEI price: 0.xx
- USDC price: 1.00
- OraclePriceDecimal (for both SEI and USDC): 8

- State variable oracleDecimals: 36
- SEI token decimal: 18
- USDC token decimal: 6

The "`getPrice`" function returns:

$$\frac{0.xx \times 10^{6+36-18}}{1.00}$$

The result simplifies to:

$$0.xx \times 10^{24}$$

When used in the "`getTokenValue`" function, if the user supplies 1 wei of sei token, the asset token value is calculated as:

$$1 \times 0.xx \times 10^{24} \div 10^{36} = 0$$

In this case, the calculated asset token value is zero which is lost the precision.

2. Precision loss

The protocol asset value calculation introduces precision loss.

The second issue lies in the "`getPrice`" function's else code branch is critical. In this case, if both assetPrice and basePrice use the same price decimal, the returned value may incorrectly evaluate to zero. Consider the following example.

If the asset token is WETH and the base token is USDC, consider the following parameters:

- WETH price: $3000.00
- USDC price: $1.00
- OraclePriceDecimal (for both WETH and USDC): 8
- State variable oracleDecimals: 8
- WETH token decimal: 18
- USDC token decimal: 6

The expected return value of the "`getPrice`" function will be calculated as:

$$\frac{3000.00 \times 10^{6} \times 10^{8} \times 10^{8}}{1.00 \times 10^{18} \times 10^{8}}$$

When a user deposits 1e18 WETH, the expected result of "`_getTokenValue`" is:

$$10^{18} \times \frac{3000.00 \times 10^6 \times 10^8 \times 10^8}{1.00 \times 10^{18} \times 10^8} \div 10^8 = 3000 \times 10^6$$

However, due to the implementation of the "`getPrice`" function, the return price is calculated based on the else branch as follows:

$$\left(3000.00 \times 10^8\right) \div 10^4 \div \left(1.00 \times 10^8\right) = 0$$

the result of "`_getTokenValue`" is:

$$1e18 \times 0 \div 1e8 = 0$$

The root cause of this issue is the improper normalization of asset value decimals. The code attempts to normalize the asset value decimals of different tokens to the "`baseTokenDecimal`"(set to "`6`" in the test) using the formula:

$$tokenAmount \times \frac{10^{baseTokenDecimal}}{10^{AssetTokenDecimal}}$$

However, this approach is flawed as it loses precision in the fractional part of the "`tokenAmount`".

A more accurate normalization method would involve scaling the asset value decimals of different tokens to a bigger unified precision, such as $1e^{36}$. Then the scaled price should be: "`raw_price * 1e(36 - assetTokendecimal)`". The formula to calculate the asset value becomes "`tokenAmount * raw_price * 1e(36 - assetTokendecimal)`". For different token decimal assets, the final calculated asset value will be based on a "`1e36`" decimal.

The same issue also exists in the "`SEINativeOracle`" contract and "`UniswapV3TwapOracle`" contract.

Consider revising the logic of normalizing the asset token decimals of different tokens to a unified precision.

Resolution

The team decided to resolve this issue by rounding up the asset value in "`_getTokenValue`" instead. Fixed by commit "`7015d5d`".

TEA-REX

[M-02] "_openPosition" not transfer fees to "_feeConfig.treasury"

Users should pay a "TradingFee" when opening a position through the "_openPosition" function of "TradingCore".

```
/* contracts/trading/TradingCore.sol */
249 | function _openPosition(
276 |     uint256 tradingFee = _calculateTradingFee(false, _borrowAmount, _feeConfig);
```

However, the "_openPosition" function does not transfer the "TradingFee" to "_feeConfig.treasury".

This oversight may result in the "TradingFee" for open positions being locked in the "TradingCore" contract.

To resolve this issue, consider adding the following code after line 276.

```
_collectTradingFee(debt, tradingFee, _feeConfig);
```

Resolution

Fixed by commit "c77044b".

TEA-REX

[M-03] Incorrect price check in "ClosePosition" function

In the "`_getLiquidationPrice`" function, the position's liquidation price is calculated as the "`assetPriceInDebt`", which is

$$\frac{debtAmount}{assetAmount + liquidateLossRatioThreshold * marginAmount}$$

```
/* contracts/trading/MarketNFT.sol */
function _getLiquidationPrice(
    Position memory _position,
    uint256 _debtAmount,
    uint8 _oracleDecimals
) internal view returns (uint256 price) {
    if (_position.isMarginAsset) {
        price = (10 ** _oracleDecimals).mulDiv(
            Percent.MULTIPLIER * _debtAmount,
            Percent.MULTIPLIER * _position.assetAmount + liquidateLossRatioThreshold * _position.marginAmount
        );
    }
    else {
        if (Percent.MULTIPLIER * _debtAmount <= liquidateLossRatioThreshold * _position.marginAmount) {
            price = 0;
        }
        else {
            price = (10 ** _oracleDecimals).mulDiv(
                Percent.MULTIPLIER * _debtAmount - liquidateLossRatioThreshold * _position.marginAmount,
                Percent.MULTIPLIER * _position.assetAmount
            );
        }
    }
```

In the "`closePosition`" function, the user swaps the asset token for the debt token to repay the debt. For non-liquidation mode operations, there is a check to ensure that the "`assetPriceInDebt`" is greater than the required price, when the user consumes assets to decrease the debt.

The condition "`totalConsumedAssetToken * _debtAmount >= position.swappableAmount * _decreasedDebtAmount`" is equivalent to:

$$\frac{debtAmount}{swappableAmount} >= \frac{decreasedDebtAmount}{totalConsumedAssetToken}$$

Here, $\frac{decreasedDebtAmount}{totalConsumedAssetToken}$ represents the current "`assetPriceInDebt`" in the swap operation while the $\frac{debtAmount}{swappableAmount}$ represents the check price.

The "`swappableAmount`" is a combination of the position's "`assetAmount`" and "`marginAmount`", so the price in this check becomes:

$$\frac{debtAmount}{assetAmount + marginAmount}$$

This is lower than the price calculated in the "`_getLiquidationPrice`" function.

The user can use a price equal to or lower than the position's liquidation price when performing a non-liquidation mode swap operation. This makes it possible for the user to self-liquidate and bypass the liquidation fee.

```
/* contracts/trading/MarketNFT.sol */
if (_mode != CloseMode.Liquidate) {
    uint256 totalConsumedAssetToken = _swappedAssetToken + _tradingFee;
    if (
        totalConsumedAssetToken * _debtAmount >= (
            position.isMarginAsset ?
                position.swappableAmount * _decreasedDebtAmount :
                position.swappableAmount * _decreasedDebtAmount + position.marginAmount *
↪ totalConsumedAssetToken
        )
    ) revert BadCloseRate();
}

/* contracts/trading/MarketNFT.sol */
positions[positionId] = Position({
    status: PositionStatus.Open,
    isLongToken0: _isLongToken0,
    isMarginAsset: isMarginAsset,
    stopLossRateTolerance: _stopLossRateTolerance,
    initialLeverage: leverage,
    marginAmount: _marginAmount,
    interestRateModelType: _interestRateModelType,
    borrowId: _borrowId,
    assetAmount: _assetAmount,
    swappableAmount: isMarginAsset ? _assetAmount + _marginAmount : _assetAmount,
    takeProfit: _takeProfit,
    stopLoss: _stopLoss
});
```

To resolve this issue, consider changing the "`position.swappableAmount`" in the "`closePosition`" function to "`_position.assetAmount + liquidateLossRatioThreshold * _position.marginAmount`".

Resolution

The team acknowledged this finding and decided to make no changes.

They clarified that collecting liquidation fees is not the first priority in the design, as liquidation fee is used to incentivize users to close their position actively, reduce loss and extra liquidation cost by liquidators.

TEA-REX
[M-04] "onlyNotPaused" check not implemented in "TradingCore"

In the "`TradingCore`" contract, It inherits the "`pausableupgradeable`" contract.

However, the "`onlyNotPaused`" check is not implemented in relevant functions (e.g. the "`openPosition`" function). This makes the "pause all markets" feature ineffective.

```
/* contracts/trading/TradingCore.sol */
function isAllMarketPaused() external view override returns (bool) {
    return paused();
}

/* contracts/trading/TradingCore.sol */
function openPosition(
    address _market,
    IRouter.InterestRateModelType _interestRateModelType,
    ERC20PermitUpgradeable _longTarget,
    uint256 _marginAmount,
    uint256 _borrowAmount,
    uint256 _minAssetAmount,
    uint256 _takeProfit,
    uint256 _stopLoss,
    uint24 _stopLossRateTolerance,
    address _swapRouter,
    bytes calldata _data
) external override nonReentrant returns

/* contracts/trading/TradingCore.sol */
function closePosition(
    address _market,
    uint256 _positionId,
    uint256 _assetTokenToSwap,
    uint256 _minDecreasedDebtAmount,
    ICalldataProcessor _calldataProcessor,
    address _swapRouter,
    bytes calldata _data
) external override nonReentrant returns
```

Consider adding the relevant pause check in the "`TradingCore`" contract.

Resolution

Fixed by commit "`e4e9437`".

TEA-REX

[M-05] Incorrect TWAP price formula

According to the implementation of the price oracle in the "`ChainlinkOracle`" and "`SEINativeOracle`", the TWAP price formula is

$$AssetPrice * \frac{10^{DECIMALS+baseTokenDecimal}}{10^{AssetTokenDecimal}}$$

However, in the "`UniswapV3TwapOracle`" contract, the "`getPrice`" function returns the price value as: "`priceX192.mulDiv(10 ** (DECIMALS + _poolInfo.decimals0 - _poolInfo.decimals1)`" when the asset token is "`poolInfo.decimals0`".

The term "`_poolInfo.decimals0 - _poolInfo.decimals1`" normalizes the difference in decimals between "`token0`" and "`token1`", ensuring that the price can be calculated based on the ratio of reserves for tokens with different decimals. Thus, the price value formula is

$$AssetPrice * 10^{DECIMALS}$$

which is inconsistent and incorrect compared to the TWAP formula used in the other oracles.

Example:

Assume the oracle pool is "`WETH (asset) : USDC (baseAsset)`" and the "`WETH`" price is $3000. Consider the following parameters:

- token0: WETH
- token1: USDC
- token0 Reserve: 1e18 wei
- token1 Reserve: 3000 * 1e6 wei

The "`getPrice`" function in the "`UniswapV3TwapOracle`" returns:

$$3000 * 10^{DECIMALS}$$

The "`getPrice`" function in the "`SEINativeOracle`" returns:

$$3000 * 10^{mulDecimals-divDecimals}$$

which is

$$3000 * \frac{10^{DECIMALS+baseTokenDecimal}}{10^{assetTokenDecimal}}$$

```solidity
/* contracts/trading/ChainlinkOracle.sol */
function getPrice(address _asset) external view returns (uint256 price) {
    OracleInfo storage assetInfo = oracleInfo[IERC20Metadata(_asset)];
    require(address(assetInfo.priceOracle) != address(0), AssetNotEnabled());
    if (_asset == address(baseAsset)) {
        return 10 ** priceDecimals;
    }
    OracleInfo storage baseInfo = oracleInfo[baseAsset];
    (,int256 assetPrice,,,) = assetInfo.priceOracle.latestRoundData();
    (,int256 basePrice,,,) = baseInfo.priceOracle.latestRoundData();
    uint256 mulDecimals = baseInfo.totalDecimals + priceDecimals;
    uint256 divDecimals = assetInfo.totalDecimals;
    if (mulDecimals > divDecimals) {
        price = Math.mulDiv(uint256(assetPrice), 10 ** (mulDecimals - divDecimals), uint256(basePrice));
    }
    else {
        price = uint256(assetPrice) / (10 ** (divDecimals - mulDecimals)) / uint256(basePrice);
    }
}

/* contracts/trading/UniswapV3TwapOracle.sol */
function getPrice(address _asset) external override view returns (uint256 price) {
    PoolInfo[] memory _poolInfoChain = poolInfoChain[_asset];
    if (_poolInfoChain.length == 0) revert AssetNotEnabled();
    price = 10 ** DECIMALS;
    if (address(_poolInfoChain[0].pool) == address(1)) {
        return price;
    }
    uint256 relativePrice;
    uint32[] memory secondsAgo = new uint32[](2);
    int56[] memory tickCumulatives;
    uint256 sqrtPriceX96;
    PoolInfo memory _poolInfo;
    for (uint256 i; i < _poolInfoChain.length; ) {
        _poolInfo = _poolInfoChain[i];
        secondsAgo[0] = _poolInfo.twapInterval;
        (tickCumulatives, ) = _poolInfo.pool.observe(secondsAgo);
        sqrtPriceX96 = TickMath.getSqrtRatioAtTick(
            int24((tickCumulatives[1] - tickCumulatives[0]) / int64(uint64(_poolInfo.twapInterval)))
        );
        if (_poolInfo.assetIsToken0) {
            if (sqrtPriceX96 <= type(uint128).max) {
                uint256 priceX192 = uint256(sqrtPriceX96) * sqrtPriceX96;
                relativePrice = _poolInfo.decimals0 > _poolInfo.decimals1 ?
                    priceX192.mulDiv(10 ** (DECIMALS + _poolInfo.decimals0 - _poolInfo.decimals1), POW_2_192) :
                    priceX192.mulDiv(10 ** DECIMALS, POW_2_192) / (10 ** (_poolInfo.decimals1 - _poolInfo.decimals0));
            }
            else {
                uint256 priceX128 = sqrtPriceX96.mulDiv(sqrtPriceX96, POW_2_64);
                relativePrice = _poolInfo.decimals0 > _poolInfo.decimals1 ?
                    priceX128.mulDiv(10 ** (DECIMALS + _poolInfo.decimals0 - _poolInfo.decimals1),
↪ POW_2_128) :
                    priceX128.mulDiv(10 ** DECIMALS, POW_2_128) / (10 ** (_poolInfo.decimals1 -
↪ _poolInfo.decimals0));
            }
        }
        else {
            relativePrice = _poolInfo.decimals1 > _poolInfo.decimals0 ?
                POW_2_192.mulDiv(10 ** DECIMALS, sqrtPriceX96).mulDiv(
                    10 ** (_poolInfo.decimals1 - _poolInfo.decimals0),
                    sqrtPriceX96
                ) :
                POW_2_192.mulDiv(
                    10 ** DECIMALS,
                    sqrtPriceX96
                ) / 10 ** (_poolInfo.decimals0 - _poolInfo.decimals1) / sqrtPriceX96;
        }
        price = price.mulDiv(relativePrice, 10 ** DECIMALS);
        unchecked { i = i + 1; }
    }
}
```

Consider updating the price formula in the "`UniswapV3TwapOracle`" contract.

Resolution

Fixed by commit "`afa28c7`".

TEA-REX

[L-01] Unsafe cast in "ChainlinkOracle"

In the "`ChainlinkOracle`" contract, the price returned by "`.priceOracle.latestRoundData()`" is directly converted to "`uint256`".

However, because this value is originally of type "`int256`" and may be negative, this unsafe cast will not revert but return a very large number (the complement representation of 256 bits) instead.

```
/* contracts/trading/ChainlinkOracle.sol */
function getPrice(address _asset) external view returns (uint256 price) {
    (,int256 assetPrice,,,) = assetInfo.priceOracle.latestRoundData();
    (,int256 basePrice,,,) = baseInfo.priceOracle.latestRoundData();
    if (mulDecimals > divDecimals) {
        price = Math.mulDiv(uint256(assetPrice), 10 ** (mulDecimals - divDecimals),
↪ uint256(basePrice));
    }
    else {
        price = uint256(assetPrice) / (10 ** (divDecimals - mulDecimals)) / uint256(basePrice);
    }
}
```

Consider reverting if a negative value is received from Chainlink. For example, add the following checks

```
error OraclePriceIsInvalid();
if (assetPrice < 0) revert OraclePriceIsInvalid();
if (basePrice < 0) revert OraclePriceIsInvalid();
```

This ensures that any negative value is flagged and the transaction reverts, preventing potential misinterpretations or incorrect operations caused by unsafe casting.

Resolution

Fixed by commit "`68516ea`".

TEA-REX

[L-02] Incorrect rounding direction in function "commitBorrow"

In the function "`commitBorrow`", the calculated "`_borrowedTeaToken`" should be rounded up instead of being rounded down to ensure the rounding direction benefits the protocol.

```
/* contracts/lending/Pool.sol */
uint256 borrowedTeaTokenAmount = _toTeaToken(_amountToBorrow, newBorrowedConversionRate, false);
id = idCounter;
idCounter = idCounter + 1;
debtInfo[id] = DebtInfo({
    isClosed: false,
    borrowedTeaToken: borrowedTeaTokenAmount
});
borrowedTeaToken = borrowedTeaToken + borrowedTeaTokenAmount;
emit Borrowed(_account, id, _amountToBorrow, borrowedTeaTokenAmount);

/* contracts/lending/Pool.sol */
function _toTeaToken(
    uint256 _underlyingAmount,
    uint256 _conversionRate,
    bool _isRoudingUp
) internal view returns (
    uint256 teaTokenAmount
) {
    teaTokenAmount = _isRoudingUp ?
        _underlyingAmount.mulDiv(RATE_MULTIPLIER * DECIMALS_MULTIPLIER, _conversionRate) :
        _underlyingAmount.mulDiv(RATE_MULTIPLIER * DECIMALS_MULTIPLIER, _conversionRate,
↪ Math.Rounding.Ceil);
}
```

It's recommended to replace

```
_toTeaToken(_amountToBorrow, newBorrowedConversionRate, false)
```

with

```
_toTeaToken(_amountToBorrow, newBorrowedConversionRate, true)
```

Resolution

Fixed by commit "`0b35b60`".

TEA-REX

[L-03] Interest fee not collected when updating reserve ratio

According to the protocol design, the underlying asset reserve ratio affects the asset utilization rate, which subsequently influences the interest rate.

However, in the "`setReserveRatio`" function, the interest fee is not collected before updating the asset reserve ratio. This oversight results in past asset interest being calculated based on an inaccurate interest rate.

```
/* contracts/lending/Pool.sol */
uint256 interestRate = IInterestRateModel(router.getInterestRateModel(interestRateModelType)).getBorrowRate(
    suppliedUnderlying,
    borrowedUnderlying,
    reserveRatio
);
uint256 rateDeltaInterest = _calculateInterests(_borrowedConversionRate, interestRate, timeElapsed);
uint256 rateDeltaFee = _calculateInterests(_borrowedConversionRate, router.getFeeConfig().borrowFee, timeElapsed);
interest = _borrowedTeaToken.mulDiv(rateDeltaInterest, RATE_MULTIPLIER);

/* contracts/lending/Pool.sol */
function setReserveRatio(uint24 _ratio) external override onlyOwner {
    _setReserveRatio(_ratio);
}

/* contracts/lending/Pool.sol */
function _setReserveRatio(uint24 _ratio) internal {
    if (_ratio >= Percent.MULTIPLIER) revert InvalidPercentage();
    reserveRatio = _ratio;
}
```

It's recommended to collect the interest fee before updating the asset reserve ratio.

Resolution

Fixed by commit "`e7ece19`".

TEA-REX

[L-04] Protocol fee not collected during "FeeConfig" update

In the “`setFeeConfig`” function, the protocol fee is not collected before updating the asset “`_borrowFee`”.

This oversight causes the past asset protocol fee to be calculated based on an “`_borrowFee`” rate.

```
/* contracts/lending/Router.sol */
function setFeeConfig(address _treasury, uint32 _borrowFee, uint32 _withdrawalFee) external override onlyOwner {
    if (_borrowFee > FEE_CAP) revert ExceedsFeeCap();
    if (_withdrawalFee > FEE_CAP) revert ExceedsFeeCap();
    feeConfig = FeeConfig({ treasury: _treasury, borrowFee: _borrowFee, withdrawalFee: _withdrawalFee });
    emit FeeConfigSet(msg.sender, _treasury, _borrowFee);
}

/* contracts/lending/Pool.sol */
uint256 rateDeltaFee = _calculateInterests(_borrowedConversionRate, router.getFeeConfig().borrowFee, timeElapsed);
interest = _borrowedTeaToken.mulDiv(rateDeltaInterest, RATE_MULTIPLIER);
fee = _borrowedTeaToken.mulDiv(rateDeltaFee, RATE_MULTIPLIER);

/* contracts/lending/Pool.sol */
(interest, fee, newSuppliedConversionRate, newBorrowedConversionRate) = _collectInterestAndFee();
lastAccumulateTimestamp = block.timestamp;
suppliedConversionRate = newSuppliedConversionRate;
borrowedConversionRate = newBorrowedConversionRate;
if (fee > 0) {
    pendingFee = pendingFee + fee;
    emit BorrowFeeAccumulated(block.timestamp, fee);
}
```

It’s recommended to collect the protocol fee before updating the asset “`_borrowFee`” ratio.

Resolution

The team acknowledged this finding and clarified that it’s not practical to enumerate all pools and collect fee in a single transaction. The team will run a script to collect fees for all pools before setting new fee structure to mitigate the effect.

TEA-REX

[L-05] Inaccurate liquidatable check in "_ensureNotLiquidated"

The function "`_ensureNotLiquidated`" requires the latest "`lossRatio`" to be **less than or equal** to the "`liquidateLossRatioThreshold`" to ensure the position cannot be liquidated.

However, in the relevant liquidation price check, only "`assetPriceInDebt`" is **less than or equal** to the "`liquidationPrice`" (which is equivalent to the "`lossRatio`" being **greater than or equal** to the "`liquidateLossRatioThreshold`") is checked.

Therefore, when the position's "`lossRatio`" is exactly equal to the "`liquidateLossRatioThreshold`", the position would still face liquidation.

```
/* contracts/trading/MarketNFT.sol */
else if (_ensureNotLiquidated) {
    uint256 debtValue = _getTokenValue(oracleDecimals, newDebtAmount, debtPrice);
    uint256 assetValue = _getTokenValue(oracleDecimals, _position.assetAmount, assetPrice);
    uint256 marginValue = _getTokenValue(oracleDecimals, _position.marginAmount, marginPrice);
    uint256 lossRatio = _calculateLossRatio(marginValue, assetValue, debtValue);
    if (lossRatio > liquidateLossRatioThreshold) revert HighLossRatio();
}

/* contracts/trading/MarketNFT.sol */
else if (_mode == CloseMode.Liquidate) {
    uint256 assetPriceInDebt = _getRelativePrice(assetPrice, debtPrice, oracleDecimals);
    if (assetPriceInDebt > _getLiquidationPrice(position, _debtAmount, oracleDecimals)) revert
↪  PassivelyCloseConditionNotMet();
    if (
        _decreasedDebtAmount < _assetToDebtAmount(_swappedAssetToken,
↪  _liquidateAuctionPrice(assetPrice, debtPrice, oracleDecimals), oracleDecimals)
    ) revert WorsePrice();
}
```

Consider replacing ">" with ">=" in the "`_ensureNotLiquidated()`":

```
if (lossRatio > liquidateLossRatioThreshold) revert HighLossRatio();
```

Resolution

Fixed by commit "`4c1d9f5`".

TEA-REX

[L-06] Potential overflow in TWAP oracle

In the “`getPrice`” function of the “`UniswapV3TwapOracle`” contract, when the asset token is “`poolInfo.token1`”, the price calculation is performed as follows:

```
/* contracts/trading/UniswapV3TwapOracle.sol */
POW_2_192.mulDiv(10 ** DECIMALS, sqrtPriceX96).mulDiv(
    10 ** (_poolInfo.decimals1 - _poolInfo.decimals0),
    sqrtPriceX96
) :
```

Here:

- “`POW_2_192`” is “`2192`”.
- “`10 ** DECIMALS`” is “`1036`”.
- “`sqrtPriceX96`” is in $[4295128739, 1461446703485210103287273052203988822378723970342]$.

The term “`POW_2_192.mulDiv(10 ** DECIMALS, sqrtPriceX96)`” results in a value range of

$$[\frac{2^{192} * 10^{36}}{1461446703485210103287273052203988822378723970342}, \frac{2^{192} * 10^{36}}{4295128739}]$$

Simplifying further, this produces a range of [“`4e45`”, “`1e84`”]. The upper bound “`1e84`” is greater than the “`type(uint256).max`”, introducing a risk of overflow.

Resolution

The team acknowledged this finding and plans to continue to monitor prices to mitigate the impact.

TEA-REX

[L-07] Price update time not checked

In the function "`getPrice`", there is no check on the "`updatedAt`" value returned by the "`latestRoundData`" function, which could result in fetching a stale price. The same issue also exists in the "`SEINativeOracle`" contract.

```
/* contracts/trading/ChainlinkOracle.sol */
function getPrice(address _asset) external view returns (uint256 price) {
    OracleInfo storage assetInfo = oracleInfo[IERC20Metadata(_asset)];
    require(address(assetInfo.priceOracle) != address(0), AssetNotEnabled());
    if (_asset == address(baseAsset)) {
        return 10 ** priceDecimals;
    }
    OracleInfo storage baseInfo = oracleInfo[baseAsset];
    (,int256 assetPrice,,,) = assetInfo.priceOracle.latestRoundData();
    (,int256 basePrice,,,) = baseInfo.priceOracle.latestRoundData();
    uint256 mulDecimals = baseInfo.totalDecimals + priceDecimals;
    uint256 divDecimals = assetInfo.totalDecimals;
    if (mulDecimals > divDecimals) {
        price = Math.mulDiv(uint256(assetPrice), 10 ** (mulDecimals - divDecimals), uint256(basePrice));
    }
    else {
        price = uint256(assetPrice) / (10 ** (divDecimals - mulDecimals)) / uint256(basePrice);
    }
}

/* contracts/trading/SEINativeOracle.sol */
function getPrice(address _asset) external view returns (uint256 price) {
    OracleInfo memory assetInfo = oracleInfo[IERC20Metadata(_asset)];
    OracleInfo memory baseInfo = oracleInfo[baseAsset];
    require(assetInfo.nameHash != bytes32(0), AssetNotEnabled());
    if (_asset == address(baseAsset)) {
        return 10 ** priceDecimals;
    }
    uint256 basePrice;
    uint256 assetPrice;
    if (lookbackSeconds == 0) {
        // use spot price
        IOracle.DenomOracleExchangeRatePair[] memory results = ORACLE_CONTRACT.getExchangeRates();

        for (uint256 i = 0; i < results.length; i++) {
            if (bytes(results[i].denom).length == assetInfo.nameLength && keccak256(bytes(results[i].denom)) == assetInfo.nameHash) {
                assetPrice = _decodePrice(results[i].oracleExchangeRateVal.exchangeRate);
            }
            else if (bytes(results[i].denom).length == baseInfo.nameLength && keccak256(bytes(results[i].denom)) == baseInfo.nameHash) {
                basePrice = _decodePrice(results[i].oracleExchangeRateVal.exchangeRate);
            }
        }
    }
    else {
        // use twap price
        IOracle.OracleTwap[] memory results = ORACLE_CONTRACT.getOracleTwaps(lookbackSeconds);

        for (uint256 i = 0; i < results.length; i++) {
            if (bytes(results[i].denom).length == assetInfo.nameLength &&
↪ keccak256(bytes(results[i].denom)) == assetInfo.nameHash) {
                assetPrice = _decodePrice(results[i].twap);
            }
            else if (bytes(results[i].denom).length == baseInfo.nameLength &&
↪ keccak256(bytes(results[i].denom)) == baseInfo.nameHash) {
                basePrice = _decodePrice(results[i].twap);
            }
        }
    }
    if (basePrice == 0 || assetPrice == 0) {
        revert UnknownToken();
    }
    uint256 mulDecimals = baseInfo.assetDecimals + priceDecimals;
    uint256 divDecimals = assetInfo.assetDecimals;
    if (mulDecimals > divDecimals) {
        price = Math.mulDiv(assetPrice, 10 ** (mulDecimals - divDecimals), basePrice);
    }
    else {
        price = assetPrice / (10 ** (divDecimals - mulDecimals)) / basePrice;
    }
}
```

Consider adding a check for the price update time in the "`getPrice`" function.

Resolution

Fixed by commits "`4e6e127`" and "`1f8f241`".

TEA-REX

[L-08] Check "leverage" in "closePosition"

When opening positions, the "`lossRatio`" and "`leverage`" of user positions are checked, ensuring that the "`lossRatio`" is not greater than the "`openPositionLossRatioThreshold`" and the "`leverage`" does not exceed "`maxLeverage`".

When closing positions, a position can be partially closed. The handler checks the remaining position's "`lossRatio`". However, it does not check the "`leverage`" and the remaining position's "`leverage`" may exceed "`maxLeverage`".

```
/* contracts/trading/MarketNFT.sol */
    uint256 debtValue = _getTokenValue(oracleDecimals, _debtAmount, debtPrice);
    uint256 assetValue = _getTokenValue(oracleDecimals, _assetAmount, assetPrice);
    uint256 marginValue = _getTokenValue(oracleDecimals, _marginAmount, marginPrice);

    uint256 lossRatio = _calculateLossRatio(marginValue, assetValue, debtValue);
    if (lossRatio > openPositionLossRatioThreshold) revert HighLossRatio();
    uint32 leverage = debtValue.mulDiv(Percent.MULTIPLIER, marginValue).toUint32();
    if (leverage > maxLeverage) revert InvalidLeverage();

/* contracts/trading/MarketNFT.sol */
    else if (_ensureNotLiquidated) {
        uint256 debtValue = _getTokenValue(oracleDecimals, newDebtAmount, debtPrice);
        uint256 assetValue = _getTokenValue(oracleDecimals, _position.assetAmount, assetPrice);
        uint256 marginValue = _getTokenValue(oracleDecimals, _position.marginAmount, marginPrice);
        uint256 lossRatio = _calculateLossRatio(marginValue, assetValue, debtValue);
        if (lossRatio >= liquidateLossRatioThreshold) revert HighLossRatio();
    }
```

Consider adding a check for the price update time in the "`getPrice`" function.

Resolution

Fixed by commit "`2958d82`".

TEA-REX

[I-01] Code style optimization

1. Comment out the development dependency library

The hardhat console library should only be used during development and should not appear in the production environment. It is recommended to comment out the following code.

```
/* contracts/lending/VariableInterestRateModel.sol */
010 | import "hardhat/console.sol";
```

2. Remove extra blank lines

To clean up the code, remove the unexpected blank line at line 138 in the "Pool" contract.

```
/* contracts/lending/Pool.sol */
128 | function _toUnderlying(
137 |     _teaTokenAmount.mulDiv(_conversionRate, RATE_MULTIPLIER * DECIMALS_MULTIPLIER, Math.Rounding.Ceil);
138 |
139 | }
```

3. Reorganize the Pool.claimFee function

Consider changing "Pool.claimFee" function from

```
/* contracts/lending/Pool.sol */
232 | function claimFee() external override nonReentrant onlyNotPaused returns (uint256 claimedFee, uint256 unclaimedFee) {
233 |     _collectInterestFeeAndCommit();
234 |
235 |     ERC20PermitUpgradeable _underlyingAsset = underlyingAsset;
236 |     uint256 balance = _underlyingAsset.balanceOf(address(this));
237 |     unclaimedFee = pendingFee;
238 |     claimedFee = balance > unclaimedFee ? unclaimedFee : balance;
239 |
240 |     if (claimedFee > 0) {
241 |         unclaimedFee = unclaimedFee - claimedFee;
242 |         pendingFee = unclaimedFee;
243 |         address treasury = router.getFeeConfig().treasury;
244 |         _underlyingAsset.safeTransfer(treasury, claimedFee);
245 |
246 |         emit FeeClaimed(treasury, claimedFee);
247 |     }
248 | }
```

to

```
function claimFee() external override nonReentrant onlyNotPaused returns (uint256 claimedFee, uint256
↪ unclaimedFee) {
    _collectInterestFeeAndCommit();
    uint256 balance = underlyingAsset.balanceOf(address(this));
    claimedFee = balance > pendingFee ? pendingFee : balance;
    if (claimedFee > 0) {
        pendingFee -= claimedFee;
        address treasury = router.getFeeConfig().treasury;
        underlyingAsset.safeTransfer(treasury, claimedFee);
        emit FeeClaimed(treasury, claimedFee);
    }
    unclaimedFee = pendingFee;
}
```

- Remove unnecessary variable "`_underlyingAsset`" and use "`underlyingAsset`" instead.
- Place the "`unclaimedFee`" assignment at the end to avoid duplicate updates and logical jumps.
- Use "`pendingFee -= claimedFee`" instead of explicit subtraction operation.

4. Function naming

Consider changing the function name from "`_updateassetTokenToSwap`" to "`_updateAssetTokenToSwap`" to align with standard camelCase naming conventions.

5. Optimize loop statements to save gas

Compared to "`i++`" or "`i--`", using "`++i`" or "`--i`" is more gas-efficient. Furthermore, in scenarios where overflow is not a concern, wrapping the operation in an "`unchecked`" block can achieve additional gas savings.

Take the "`setWhitelistedOperator`" function in the "`TradingCore`" contract as an example.

```
for (uint256 i; i < length; i = i + 1) {
    whitelistedOperator[_accounts[i]] = _isWhitelisted[i];
}
```

can be changed to:

```
for (uint256 i; i < length;) {
    whitelistedOperator[_accounts[i]] = _isWhitelisted[i];
    unchecked {
        ++i;
    }
}
```

6. Typo

The "`_isRoudingUp` "should be "`_isRoundingUp`".

```
/* contracts/lending/Pool.sol */
function _toUnderlying(
    uint256 _teaTokenAmount,
    uint256 _conversionRate,
    bool _isRoudingUp
) internal view returns (
    uint256 underlyingAmount
) {
    underlyingAmount = _isRoudingUp ?
        _teaTokenAmount.mulDiv(_conversionRate, RATE_MULTIPLIER * DECIMALS_MULTIPLIER) :
        _teaTokenAmount.mulDiv(_conversionRate, RATE_MULTIPLIER * DECIMALS_MULTIPLIER, Math.Rounding.Ceil);
}

function _toTeaToken(
    uint256 _underlyingAmount,
    uint256 _conversionRate,
    bool _isRoudingUp
) internal view returns (
    uint256 teaTokenAmount
) {
    teaTokenAmount = _isRoudingUp ?
        _underlyingAmount.mulDiv(RATE_MULTIPLIER * DECIMALS_MULTIPLIER, _conversionRate) :
        _underlyingAmount.mulDiv(RATE_MULTIPLIER * DECIMALS_MULTIPLIER, _conversionRate, Math.Rounding.Ceil);
}

```

7. Use modifier instead of repetitive checks

The "`Router`" contract enforces that only the "`TradingCore`" contract is authorized to call the "`borrow`", "`commitBorrow`", and "`repay`" functions.

```
/* contracts/lending/Router.sol */
if (msg.sender != tradingCore) revert CallerIsNotTradingCore();
/* contracts/lending/Router.sol */
if (msg.sender != tradingCore) revert CallerIsNotTradingCore();
```

```
/* contracts/lending/Router.sol */
if (msg.sender != tradingCore) revert CallerIsNotTradingCore();
```

Consider integrating this requirement into a modifier named "`onlyTradingCore`".

Resolution

Fixed by commit "`05c7538`".

TEA-REX

[I-02] The "repay" function does not check if the "debtInfo" is already closed

In the "`Pool`" contract, the "`isClosed`" flag in the "`debtInfo`" structure is intended to indicate the end of a debt.

However, the "`repay`" function does not verify whether "`debtInfo.isClosed`" is "`True`" before proceeding. This oversight could allow the same debtInfo to be repaid and marked as closed multiple times.

Although there is a status check in the "`closePosition` "function, the current implementation of the Pool contract is not secure. This weakness cannot be exploited because the caller ("`MarketNFT`") happens to check the position status before calling the "`repay`" function.

```
/* contracts/trading/MarketNFT.sol */
391 | if (position.status != PositionStatus.Open) revert InvalidPositionStatus();
```

However, if other methods in the future call "`router.repay`", this flaw may become exploitable.

Consider adding the following check in the "`repay`" function:

```
DebtInfo memory _debtInfo = debtInfo[_id];
if (_debtInfo.isClosed) revert DebtPositionIsClosed();
```

Resolution

Fixed by commit "`dc84ecb`".

TEA-REX

[I-03] Validate parameters in "setSupplyCap" and "setBorrowCap"

“setSupplyCap()” and “setBorrowCap()” do not check if the “_borrowCap” is larger than “_supplyCap”.

Consider adding the following constraints in the “setSupplyCap” and “setBorrowCap” functions.

```
if (_borrowCap > _supplyCap) revert InvalidCap();
```

Resolution

Fixed by commit “8be8fec”.

TEA-REX

[I-04] Check loss ratio in function "addMargin"

Users can add margins to their positions via the "`addMargin()`" to mitigate the risk of liquidation.

However, the function lacks a "`LossRatio`" check to verify that the position remains healthy after the user deposits margin tokens.

```
/* contracts/trading/MarketNFT.sol */
function addMargin(
    uint256 _positionId,
    uint256 _addedAmount
) external override nonReentrant onlyNotPaused onlyTradingCore {
    Position storage position = positions[_positionId];
    if (position.status != PositionStatus.Open) revert InvalidPositionStatus();
    position.marginAmount = position.marginAmount + _addedAmount;
}
```

Consider adding the loss ratio check in the function "`addMargin`".

Resolution

The team acknowledged this finding.

TEA-REX

[I-05] Avoid using user-provided "calldataProcessor" address

In the "`closePosition`" function, the user-passed "`calldataProcessor`" is used to process the swap data. However, it's not sufficiently validated.

```
/* contracts/trading/TradingCore.sol */
function _closePosition(
    ICalldataProcessor _calldataProcessor,
) internal returns (
) {
    if (address(_calldataProcessor) != address(0)) {
        _data = _calldataProcessor.processCalldata(
            _router.debtOfUnderlying(debt, position.interestRateModelType, position.borrowId),
            _data
        );
    }
```

It is recommended to use a state variable to store the valid "`calldataProcessor`" address instead of relying on untrusted external input.

Resolution

The team acknowledged this finding.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Be An Inc dba Teahouse (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

Sec3™