Centuari Audit Report

Version 1.0

KasturiSec

July 8, 2025

Centuari Audit Report

KasturiSec

July 8, 2025

Prepared by: KasturiSec

Auditors:

- JuggerNaut63
- 0xDemon
- farismaulana

Table of Contents

About

KasturiSec consists of many best smart contract security researchers in the space. Although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @KasturiSec.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About Centuari

Centuari is an innovative decentralized lending protocol powered by a deCentralized Lending Order Book (CLOB) system. It enables both retail and institutional users to access fixed-rate loans.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - leads to a moderate material loss of assets in the protocol or moderately harms a group of users.
- Low - leads to a minor material loss of assets in the protocol or harms a small group of users.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Action required for severity levels

- High - Must fix (before deployment if not already deployed)

- Medium - Should fix
- Low - Could fix

Audit Details

Scope

The review conducted from 24 June to 27 June 2025.

Review commit hashes: 2524432f7f541308ca5bc5e7bff9078f1dad4bf7

Fixed commit hashes:

- cc059779c66ebb88e5a7c4268f2d83897ab5fde3
- 99ddd87377975b58e06ce9c7f10dfe37f1755f73

```
src/core/CentuariBondToken.sol
src/core/centuari-clob/CentuariCLOBMarketManager.sol
src/core/centuari-clob/CentuariCLOBMarket.sol
src/core/centuari-clob/CentuariCLOBOrderGroupManager.sol
src/core/centuari-clob/CentuariCLOB.sol
src/core/Centuari.sol
src/core/DataStore.sol
src/interfaces/centuari-clob/ICentuariCLOBMarket.sol
src/interfaces/centuari-clob/ICentuariCLOB.sol
src/interfaces/ICentuariCallbacks.sol
src/interfaces/ICentuari.sol
src/interfaces/IDataStore.sol
src/interfaces/IOracleManager.sol
src/libraries/centuari/CentuariDSLib.sol
src/libraries/centuari/CentuariErrorsLib.sol
src/libraries/centuari/CentuariEventsLib.sol
src/libraries/centuari-clob/CentuariCLOBDSLib.sol
src/libraries/centuari-clob/CentuariCLOBErrorsLib.sol
src/libraries/centuari-clob/CentuariCLOBEventsLib.sol
src/libraries/centuari-clob/CentuariCLOBOrderProcessorLib.sol
src/libraries/centuari-clob/CentuariCLOBValidator.sol
src/libraries/centuari-clob/OrderQueueLib.sol
src/libraries/MarketConfigLib.sol
src/types/CommonTypes.sol
```

Roles

- **Owner/Admin**: The owner of the contract, can perform administrative actions such as pausing the contract, setting parameters, etc.

- **User**: Users who can trade on the platform.

Executive Summary

Over the course of security review, we found a total of 16 issues, categorized as follows:

ID	Title	Severity	Status
[H-01]	`Centuari::addAnotherCollateral` does not transfer the collateral added from borrower	High	Resolved
[H-02]	Unhealthy Positions Cannot Be Liquidated Before Maturity	High	Resolved
[H-03]	`CENTUARI::withdrawCollateral` can be used to withdraw all collateral even if borrower still have debt	High	Resolved
[H-04]	Logic flaw in the `findMatchOrder()` function when the `oppositeOrderGroup` has `amount == 0` or status is `CANCELED`	High	Resolved
[H-05]	Lender's order that is not matched cant be cancelled	High	Resolved
[M-01]	No fee receiver set	Medium	Resolved
[M-02]	Users Can Liquidate Themselves Without Penalty	Medium	Resolved
[M-03]	Rate calculation disregarding the maturity days	Medium	Resolved
[M-04]	`Centuari::supplyCollateralAndBorrow` borrower health check only account amount without interest	Medium	Resolved
[M-05]	Partially Filled Market Orders Incorrectly Added to Queue	Medium	Acknowledged
[M-06]	if `placeMarketOrder` result in no match, would cause user fund stuck	Medium	Resolved
[M-07]	Incorrect fee handling when match order happening	Medium	Resolved
[L-01]	`CentuariCLOBMarketManager::deactivateMarket` is missing check if the current market still have user funds	Low	Resolved

ID	Title	Severity	Status
[L-02]	`CentuariCLOBMarketManager::deactivateMarket` always deactivate all maturity of given market config	Low	Acknowledged
[L-03]	Inconsistent use of denominator when calculating fees	Low	Resolved
[L-04]	Invalid denominator used when creating LLTV metadata on CBT	Low	Resolved

Findings

High

[H-01] `Centuari::addAnotherCollateral` does not transfer the collateral added from borrower

Severity

Impact: High

Likelihood: High

Description

`addAnotherCollateral()` function in the `Centuari` contract had an issue that allowed users to add "phantom collateral" to their positions without actually transferring collateral tokens to the contract. This function simply updated the internal storage without verifying or transferring actual tokens, creating a mismatch between the storage data and the assets actually held by the contract.

The problem is Incomplete implementation of `addAnotherCollateral()` function in Centuari.sol#L260-L276.

```
function addAnotherCollateral(MarketConfig memory config, uint256
    maturity, address user, uint256 amount)
     external
     nonReentrant
     onlyActiveMarket(config.marketId())
     onlyActiveMaturity(config.marketId(), maturity)
     onlyActiveUser(config.marketId(), maturity, user)
```

```
    whenNotPaused
{
    if (amount == 0) revert CentuariErrorsLib.InvalidAmount();

    IDataStore dataStore = IDataStore(dataStores[config.marketId()]);

    // ❌ MAIN PROBLEM: Only updating storage without transferring token
    CentuariDSLib.setUserCollateral(dataStore, maturity, user,
        CentuariDSLib.getUserCollateral(dataStore, maturity, user) +
            amount
    );

    // ❌ NO TOKEN TRANSFER FROM USER TO CONTRACT!
    // MISSING: IERC20(collateralToken).safeTransferFrom(msg.sender,
        address(this), amount);

    emit CentuariEventsLib.AddAnotherCollateral(config.marketMaturityId
        (maturity), user, amount);
}
```

For comparison, the `withdrawCollateral()` function correctly `safeTransfer()`s the token to the user and the `supplyCollateralAndBorrow()` function receives the token via `CentuariCLOB` before being called. Only `addAnotherCollateral()` does not transfer the token.

Health Check in `_isHealthy()` function in `Centuari.sol` Centuari.sol#L191-L204 becomes dependent on inaccurate storage data.

```
function _isHealthy(Id id, uint256 maturity, address user, uint256
    collateral, uint256 amount) public view returns (bool) {
    // Using data from storage that may contain phantom collateral
    uint256 collateralValue = ((CentuariDSLib.getUserCollateral(
        dataStore, maturity, user) + collateral) * collateralPrice) /
        collateralDecimals;
    // Health check becomes inaccurate due to phantom collateral
}
```

Recommendation

Add token transfer from user to contract in `addAnotherCollateral()` function

```
function addAnotherCollateral(MarketConfig memory config, uint256
    maturity, address user, uint256 amount)
    external
    nonReentrant
    onlyActiveMarket(config.marketId())
    onlyActiveMaturity(config.marketId(), maturity)
    onlyActiveUser(config.marketId(), maturity, user)
```

```
    whenNotPaused
{
    // Validate that amount cannot be zero
    if (amount == 0) revert CentuariErrorsLib.InvalidAmount();

    // Validation that only the user himself can add collateral
+   if (user != msg.sender) revert CentuariErrorsLib.InvalidUser();

    // Get DataStore for the relevant market
    IDataStore dataStore = IDataStore(dataStores[config.marketId()]);

    // Get collateral token address
+   address collateralToken = dataStore.getAddress(CentuariDSLib.
    COLLATERAL_TOKEN_ADDRESS);

    // ✅ FIX: Transfer tokens from user to contract first
+   IERC20(collateralToken).safeTransferFrom(msg.sender, address(this),
     amount);

    // Adding new collateral to existing collateral
    CentuariDSLib.setUserCollateral(dataStore, maturity, user,
        CentuariDSLib.getUserCollateral(dataStore, maturity, user) +
            amount
    );

    // Emit event to record additional collateral
    emit CentuariEventsLib.AddAnotherCollateral(config.marketMaturityId
        (maturity), user, amount);
}
```

[H-02] Unhealthy Positions Cannot Be Liquidated Before Maturity

Severity

Impact: High

Likelihood: High

Description

There is a logic error in the `liquidate()` function on the `Centuari` contract that uses the wrong boolean operator. The current liquidation condition uses the `||` (OR) operator when it should use the && (AND) operator, causing unhealthy positions to not be liquidated before maturity.

The root of the problem lies in the `liquidate()` function in Centuari.sol#L360-L362.

```
function liquidate(MarketConfig memory config, uint256 maturity,
    address user) external {
     // ... other validations ...

     // ⊠ ISSUE: Use of the || operator wrong
     if ((block.timestamp < maturity) || _isHealthy(config.marketId(),
        maturity, user, 0, 0)) {
         revert CentuariErrorsLib.LiquidationNotAllowed();
     }

     // ... liquidation logic ...
}
```

The logic problem is:

Operator || (OR), liquidation is rejected if ONE of the conditions is true:

- If not yet mature = true → liquidation is rejected (even though the position is unhealthy)
- If the position is healthy = true → liquidation is rejected (this is true)

Operator && (AND) which should be, liquidation is rejected if BOTH conditions are true:

- Liquidation is only rejected if not yet mature AND the position is healthy
- If the position is unhealthy → liquidation is allowed (even though not yet mature)

Comparison:

|| (OR)

```
Not Maturity, Healthy true || true = true REJECTED ⊠TRUE
Not Maturity, Unhealthy true || false = true REJECTED ⊠WRONG
Already Maturity, Healthy false || true = true REJECTED ⊠WRONG
Already Maturity, Unhealthy false || false = false ALLOWED ⊠TRUE
```

Lines 2 and 3 give incorrect results.

&& (AND):

```
Not Maturity, Healthy true && true = true REJECTED ⊠TRUE
Not Maturity, Unhealthy true && false = false ALLOWED ⊠TRUE
Already Maturity, Healthy false && true = false ALLOWED ⊠NEED POLICY
Already Maturity, Unhealthy false && false = false ALLOWED ⊠TRUE
```

Line 3 requires a special policy (grace period or settlement mechanism).

Proof of Concept

Add to `CentuariBaseTest.t.sol`.

```
function test_LiquidationLogicError_UnhealthyPositionBeforeMaturityCannotBeLiquidated() public {
    // === SETUP BORROWING POSITION ===

    // Setup borrower with collateral and borrow
    address testBorrower = makeAddr("testBorrower");
    _mintTokensAndSetApprovals(testBorrower);

    // Test configuration
    uint256 collateralAmount = 1e18; // 1 ETH
    uint256 borrowAmount = 1800e6;   // 1800 USDC
    uint256 borrowRate = 5e16;       // 5% rate

    // Initial ETH price: $2500 (from BaseTest setup)
    // Initial health ratio: (1 ETH * $2500) / $1800 = 138% > 80% (HEALTHY)

    // Borrower performs supply collateral and borrow
    vm.prank(address(centuariCLOB));
    bool borrowSuccess = centuari.supplyCollateralAndBorrow(
        lendMarketConfigs[0],
        testMaturity,
        testBorrower,
        borrowRate,
        borrowAmount,
        collateralAmount
    );
    assertTrue(borrowSuccess, "Initial borrow should succeed");

    // Verify initial position is healthy
    bool initialHealthy = centuari._isHealthy(
        lendMarketConfigs[0].marketId(),
        testMaturity,
        testBorrower,
        0,
        0
    );
    assertTrue(initialHealthy, "Position should be initially healthy");

    // === PRICE MANIPULATION TO MAKE POSITION UNHEALTHY ===

    // Lower ETH price from $2500 to $1400
    // New health ratio: (1 ETH * $1400) / $1800 = 77% < 80% (UNHEALTHY!)

    // Get oracle for ETH/USDC
    address ethUsdcOracle = oracleManager.getOracle(
        address(mockTokens[0]), // WETH
        address(mockStableTokens[0]) // USDC
```

```
47      );
48
49      // Set new price that makes position unhealthy
50      MockOracle(ethUsdcOracle).setPrice(1400e6); // $1400 per ETH
51
52      // Verify position is now unhealthy
53      bool nowUnhealthy = centuari._isHealthy(
54          lendMarketConfigs[0].marketId(),
55          testMaturity,
56          testBorrower,
57          0,
58          0
59      );
60      assertFalse(nowUnhealthy, "Position should now be unhealthy");
61
62      // === VERIFY CONDITIONS BEFORE MATURITY ===
63
64      // Ensure it's before maturity (testMaturity = block.timestamp + 30
            days)
65      assertTrue(block.timestamp < testMaturity, "Should be before
            maturity");
66
67      // === TRY LIQUIDATION (SHOULD SUCCEED BUT WILL FAIL DUE TO BUG)
            ===
68
69      // Setup liquidator
70      address testLiquidator = makeAddr("testLiquidator");
71      _mintTokensAndSetApprovals(testLiquidator);
72
73      // Try liquidation - this should SUCCEED because position is
            unhealthy
74      // But will FAIL due to bug in liquidation logic
75      vm.prank(testLiquidator);
76      vm.expectRevert(CentuariErrorsLib.LiquidationNotAllowed.selector);
77      centuari.liquidate(lendMarketConfigs[0], testMaturity, testBorrower
            );
78
79      // === PROOF OF BUG: UNHEALTHY POSITION CANNOT BE LIQUIDATED ===
80
81      console.log("=== PROOF OF LIQUIDATION LOGIC ===");
82      console.log("Condition: Before maturity =", block.timestamp <
            testMaturity);
83      console.log("Condition: Position healthy =", !nowUnhealthy);
84
85      // === SIMULATION IF PRICE CONTINUES TO DROP ===
86
87      // Lower price again to $1000
88      MockOracle(ethUsdcOracle).setPrice(1000e6);
89
90      // Health ratio now: $1000 / $1800 = 55% (VERY UNHEALTHY!)
91      bool veryUnhealthy = centuari._isHealthy(
```

```
92          lendMarketConfigs[0].marketId(),
93          testMaturity,
94          testBorrower,
95          0,
96          0
97      );
98      assertFalse(veryUnhealthy, "Position should be very unhealthy now")
            ;
99
100     // Still cannot be liquidated due to the same bug
101     vm.prank(testLiquidator);
102     vm.expectRevert(CentuariErrorsLib.LiquidationNotAllowed.selector);
103     centuari.liquidate(lendMarketConfigs[0], testMaturity, testBorrower
            );
104 }
```

Result:

```
1  forge test --match-test
       test_LiquidationLogicError_UnhealthyPositionBeforeMaturityCannotBeLiquidated
        -vvv
2  [ ] Compiling...
3  [ ] Compiling 1 files with Solc 0.8.26
4  [ ] Solc 0.8.26 finished in 53.09s
5  Compiler run successful!
6
7  Ran 1 test for test/Centuari/CentuariBaseTest.t.sol:CentuariBaseTest
8  [PASS]
       test_LiquidationLogicError_UnhealthyPositionBeforeMaturityCannotBeLiquidated
       () (gas: 1864040)
9  Logs:
10   About to add lend orders...
11   Lend orders added successfully
12
13 Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 47.36ms
       (2.91ms CPU time)
14
15 Ran 1 test suite in 53.49ms (47.36ms CPU time): 1 tests passed, 0
       failed, 0 skipped (1 total tests)
```

Recommendation

Use && (AND) in the `liquidate()` function.

```
1  function liquidate(MarketConfig memory config, uint256 maturity,
       address user)
2      external
3      nonReentrant
4      onlyActiveMarket(config.marketId())
```

```
5       onlyActiveMaturity(config.marketId(), maturity)
6       whenNotPaused
7   {
8       if (user == address(0)) revert CentuariErrorsLib.InvalidUser();
9
10  -   if ((block.timestamp < maturity) || _isHealthy(config.marketId(),
        maturity, user, 0, 0)) {
11      // ⚠ FIX: Liquidation is only rejected if BOTH conditions are true
12      // 1. Not yet mature AND 2. Position is still healthy
13      // This allows liquidation of unhealthy positions before maturity
14  +   if ((block.timestamp < maturity) && _isHealthy(config.marketId(),
        maturity, user, 0, 0)) {
15          revert CentuariErrorsLib.LiquidationNotAllowed();
16      }
17
18      // ... other code ...
19  }
```

[H-03] `CENTUARI::withdrawCollateral` can be used to withdraw all collateral even if borrower still have debt

Severity

Impact: High

Likelihood: High

Description

lets take a look at `withdrawColalteral` in Centuari.sol#L318-L319:

```
1       function withdrawCollateral(MarketConfig memory config, uint256
           maturity, uint256 amount)
2   ...
3       {
4   ...
5           uint256 userCollateral = CentuariDSLib.getUserCollateral(
               dataStore, maturity, msg.sender);
6           if (userCollateral < amount) revert CentuariErrorsLib.
               InsufficientCollateral();
7
8   @>          CentuariDSLib.setUserCollateral(dataStore, maturity, msg.
        sender, userCollateral - amount);
9   @>          if (!_isHealthy(config.marketId(), maturity, msg.sender,
        amount, 0)) revert CentuariErrorsLib.InsufficientCollateral();
```

when withdrawing collateral for given `amount`, the function would set the user collateral to `userCollateral - amount`.

but when checking the `_isHealthy` , the collateral `amount` is also used as params. if we check the implementation of `_isHealthy`:

```
    function _isHealthy(Id id, uint256 maturity, address user, uint256
        collateral, uint256 amount) public view returns (bool) {
        IDataStore dataStore = IDataStore(dataStores[id]);

        address oracle = ORACLE_MANAGER.getOracle(dataStore.getAddress(
            CentuariDSLib.COLLATERAL_TOKEN_ADDRESS), dataStore.
            getAddress(CentuariDSLib.LOAN_TOKEN_ADDRESS));
        uint256 collateralPrice = IMockOracle(oracle).price();
        uint256 collateralDecimals = 10 ** IERC20Metadata(dataStore.
            getAddress(CentuariDSLib.COLLATERAL_TOKEN_ADDRESS)).decimals
            ();

        // borrow 1000USDC in 5% in one year, borrowed value will be
            1050USDC
        uint256 borrowedValue = CentuariDSLib.getUserBorrowValue(
            dataStore, maturity, user) + amount;
@>        uint256 collateralValue = ((CentuariDSLib.getUserCollateral(
    dataStore, maturity, user) + collateral) * collateralPrice) /
    collateralDecimals;
        uint256 maxBorrowedValue = (collateralValue * dataStore.getUint
            (CentuariDSLib.LLTV_UINT256)) / 100e16;

        return borrowedValue <= maxBorrowedValue;
    }
```

the collateral amount would later get added back and add the value of `collateralValue` and making the conditions always return true as the `borrowedValue <= maxBorrowedValue` would taken the original value like there are no withdrawal happening.

Recommendation

fix the implementation of `withdrawCollateral` :

```
        CentuariDSLib.setUserCollateral(dataStore, maturity, msg.sender
            , userCollateral - amount);
-       if (!_isHealthy(config.marketId(), maturity, msg.sender, amount
    , 0)) revert CentuariErrorsLib.InsufficientCollateral();
+       if (!_isHealthy(config.marketId(), maturity, msg.sender, 0, 0))
     revert CentuariErrorsLib.InsufficientCollateral();
```

[H-04] Logic flaw in the `findMatchOrder()` function when the `oppositeOrderGroup` has `amount == 0` or status is `CANCELED`

Severity

Impact: HIGH

Likelihood: MEDIUM

Description

When the user calls the `placeMarketOrderGroup()` / `placeLimitOrderGroup()` function, the next step is to find the matching order using the `findMatchOrder()` function.

The problem arise here, there is a check where if the value of `oppositeOrderGroupAmount == 0 || oppositeOrderGroupStatus == OrderStatus.CANCELLED` then the `oppositeOrderId` that has value or status above will be removed but in the current implementation the `OrderGroupId` that is removed from the queue is the `nextoppositeOrderId` not current `oppositeOrderId` due to a logic flaw.

The direct impact is the wrong logic match order and if the `nextoppositeOrderId` has a value that is not equal to 0 and is not in CANCELLED status, it will harm the trader who has the order group because it is removed from the queue.

Proof of Concept

The problem is in this logic block, the `oppositeOrderId` should be removed first before its value is replaced with `nextoppositeOrderId`.

```
1  if(oppositeOrderGroupAmount == 0 || oppositeOrderGroupStatus ==
       OrderStatus.CANCELLED) {
2          oppositeOrderId = nextoppositeOrderId;
3          removeOrderFromQueue(context, marketId, rate, oppositeSide,
               oppositeOrderId);
4          continue;
5      }
```

Recommendation

consider changing the logic order

```
if(oppositeOrderGroupAmount == 0 || oppositeOrderGroupStatus ==
   OrderStatus.CANCELLED) {
-       oppositeOrderId = nextoppositeOrderId;
        removeOrderFromQueue(context, marketId, rate, oppositeSide,
           oppositeOrderId);
+       oppositeOrderId = nextoppositeOrderId;
        continue;
    }
```

[H-05] Lender's order that is not matched cant be cancelled

Severity

Impact: High

Likelihood: Medium

Description

Lender does not have option to cancel their order and withdraw their loan token amount.

the `Centuari::withdraw` can only be invoked for matured Bond Token so Lender order that is not matched currently cant use this function to withdraw.

even the `CentuariCLOB::cancelOrderGroup` only delete the lender order group, not transferring the loaned amount to lender.

Recommendation

Lender should be able to withdraw loaned token amount that is already cancelled.

Medium

[M-01] No fee receiver set

Severity

Impact: Medium

Likelihood: Medium

Description

Inside the `CentuariCLOB` implementation, fee are deducted but there are no recipient for the fees.

Making the fee stuck inside the contract.

Recommendation

there are options:

1. transfer the fee each time a transaction happening
2. save the amount of fee inside a state, and add function to claim accumulated fees by owner

[M-02] Users Can Liquidate Themselves Without Penalty

Severity

Impact: Medium

Likelihood: Medium

Description

`liquidate()` function in `Centuari` contract does not have validation to prevent users from liquidating their own positions. This allows users in unhealthy positions to avoid the consequences of liquidation by performing self-liquidation, which has the same result as a normal repay + withdraw but bypasses the health check requirements. In a healthy liquidation system, there should be:

- Users in unhealthy positions are penalized
- External liquidators are incentivized/bonus
- There is an effective risk management mechanism

However, with this issue, users can avoid the liquidation penalty, get back all their collateral, and bypass the health check restrictions on withdrawals.

The root of the problem lies in the lack of identity validation, more sensible penalties and incentives in the `liquidate()` function in Centuari.sol#L352-L377.

```
1 function liquidate(MarketConfig memory config, uint256 maturity,
     address user)
2     external
```

```
3       nonReentrant
4       onlyActiveMarket(config.marketId())
5       onlyActiveMaturity(config.marketId(), maturity)
6       whenNotPaused
7   {
8       // ⛔ NO VALIDATION: msg.sender != user
9       if (user == address(0)) revert CentuariErrorsLib.InvalidUser();
10
11      // Validation of liquidation conditions
12      if ((block.timestamp < maturity) || _isHealthy(config.marketId(),
            maturity, user, 0, 0)) {
13          revert CentuariErrorsLib.LiquidationNotAllowed();
14      }
15
16      // ... liquidation logic without self-liquidation restrictions
17  }
```

Users do not lose anything when liquidated, and the liquidator does not receive any additional incentives. The liquidator pays the debt and receives collateral of equal value (1:1) and liquidation does not require a health check like withdrawal.

Recommendation

Implement a stricter penalty system, and reasonable incentives for liquidators (e.g. gas rewards).

[M-03] Rate calculation disregarding the maturity days

Severity

Impact: Medium

Likelihood: Medium

Description

lets take a look of rate calculation in function `supply` and `supplyCollateralAndBorrow`:

Centuari.sol#L221

```
1       function supply(MarketConfig memory config, uint256 maturity,
            address user, uint256 rate, uint256 amount)
2   ...
3           // mint tokenized bond to the lender
4   @>      uint256 shares = amount + ((amount * rate) / 100e16);
```

Centuari.sol#L250

```
1      function supplyCollateralAndBorrow(MarketConfig memory config,
          uint256 maturity, address user, uint256 rate, uint256 amount,
          uint256 collateral)
2  ...
3          uint256 userBorrowValue = CentuariDSLib.getUserBorrowValue(
              dataStore, maturity, user);
4  @>      uint256 additionalBorrowValue = amount + ((amount * rate) / 1
       e18);
```

here the rate would always multiplied directly into amount, there are no conversion regarding maturity.

Borrower/Lender can just create order with rate of 5% (yearly rate), and there are no difference between 1 day maturity and 90 day maturity order as the rate would directly used here.

this would cause issue where borrower would be at disadvantage as they think they borrow with 5% yearly rate but instead before maturity (for example 1 day) they need to pay 5% of their borrowed amount as interest.

for lender, this would be an advantage because of the high rate used even if the maturity is set to 1 day.

Recommendation

rate should be converted first: `convertedRate = rate * maturityDays / 365 days`

then using the `convertedRate` we can calculate the actual shares and the `additionalBorrowValue` for respective roles.

[M-04] `Centuari::supplyCollateralAndBorrow` borrower health check only account amount without interest

Severity

Impact: Medium

Likelihood: Medium

Description

lets take a look into `supplyCollateralAndBorrow` in Centuari.sol#L241-L251:

```
1  @>      if (!_isHealthy(config.marketId(), maturity, user, collateral,
       amount)) return false;
2
3          IDataStore dataStore = IDataStore(dataStores[config.marketId()
               ]);
4
5          CentuariDSLib.setUserCollateral(dataStore, maturity, user,
6              CentuariDSLib.getUserCollateral(dataStore, maturity, user)
                   + collateral
7          );
8
9  @>      uint256 userBorrowValue = CentuariDSLib.getUserBorrowValue(
       dataStore, maturity, user);
10 @>      uint256 additionalBorrowValue = amount + ((amount * rate) / 1
       e18);
11 @>      CentuariDSLib.setUserBorrowValue(dataStore, maturity, user,
       userBorrowValue + additionalBorrowValue);
```

`_isHealthy` only consider `amount` added to `original borrow amount` when checking if the borrower collateral sufficient.

but later we can see that borrower borrow value is set to `userBorrowValue + additionalBorrowValue` .

this seems fine at first but we should check that `additionalBorrowValue` are including borrower interest, effectively this set the borrow value into `original borrow amount + amount + interest` which is quite different than the check in `_isHealthy`.

it is possible after borrower matched, their order can be at higher lltv and can be liquidated

Recommendation

`_isHealthy` should also check amount + interest when calling `supplyCollateralAndBorrow` .

[M-05] Partially Filled Market Orders Incorrectly Added to Queue

Severity

Impact: Medium

Likelihood: Medium

Description

Market orders that should be executed immediately at the best available price are instead added to the queue when they experience partial fill. This violates the basic principle of market orders which should be "execute immediately or cancel" and changes its nature to a limit order without user approval.

In the `placeMarketOrderGroup()` function, when a market order is successfully partially filled, the remaining unmatched orders are added to the queue at the last successfully matched rate, instead of being canceled as it should be.

The root of the problem lies in the partially filled market order handling logic inside the `placeMarketOrderGroup()` function in CentuariCLOB.sol#L148-L149.

```
1 if(matchRate != 0 && orderGroup.status == OrderStatus.PARTIALLY_FILLED)
      {
2     // Market Orders are added to the queue if they are partially
         filled - THIS IS WRONG!
3     CentuariCLOBOrderProcessorLib.addOrderToQueue(address(market),
         marketId, msg.sender, matchRate, side, orderId, groupId);
4 }
```

While the `addOrderToQueue` function in CentuariCLOBOrderProcessorLib.sol#L184-L209 does not distinguish between market orders and limit orders.

The matching system is correct in finding counterparts, but the handling of remaining unmatched orders is wrong.

Recommendation

For market order, only execute immediately, do not queue and exit the loop as the order is completed, emit events for mismatched balances on partially filled market orders.

If the order is `PARTIALLY_FILLED`, refund part of the unmatched token amount to user.

[M-06] if `placeMarketOrder` result in no match, would cause user fund stuck

Severity

Impact: High

Likelihood: Low

Description

`CentuariCLOB::placeMarketOrder` is used when user wants to get matched with current best order in the order book. user is expected to match in this function.

but the implementation code inside CentuariCLOB.sol#L156-L157 shows that the handling of unmatched order from `placeMarketOrder` is insufficient as it only emit an event, making the funds transferred cant be withdrawn as the state are not saved in state thus the order does not have an Id and does not added into order group queue.

Recommendation

correctly handles the order if it does not match according to protocol intent (whether it revert or added into order group):

```
1          if(orderGroup.status == OrderStatus.OPEN) { //Order didnt
               matched in any market
2  -           emit CentuariCLOBEventsLib.OrderGroupNotMatchedInAnyMarket(
       orderGroup.id, orderGroup.amount, orderGroup.collateralAmount, side,
        maturity);
3  +           revert();
4          }else if(orderGroup.status == OrderStatus.PARTIALLY_FILLED) {
               //If Order Group is still partially filled, it means that
               the order has matched in at least one market and need to be
               added to order group data store
5              orderGroupManager.addOrderGroup(orderGroup, side, maturity)
                   ;
6          }
```

[M-07] Incorrect fee handling when match order happening

Severity

Impact: Medium

Likelihood: High

Description

When order is matched, for example inside the LEND block CentuariCLOBOrderProcessorLib.sol#L73-L87 the matched amount would then deducted by the corresponding fees. This is also happen in the BORROW block.

The issue is for LEND block, the borrowed amount is the amount that already deducted by the `makerFee` where the BOND token is also minted by the amount that is reduced by `takerFee`.

To explain this issue more clearly, consider a scenario:

- there exist single order (BORROW 1000e6 USDC with 5% rate)
- lender create order so amount matched = 1000e6 USDC with 5% rate
- makerFee = 1%
- takerFee = 2%

in the current implementation, on the borrower side, the contract would only acknowledged that borrower only borrow 990e6 USDC as the 10e6 USDC is for the `makerFee`.

on the lender side, the contract would only mint Bond Token by 980e6 amount, as the 20e6 USDC is for the `takerFee`.

because our simple scenario only have 1 order, we can be sure that the contract only held 1000e6 USDC that is supplied from lender.

now if we calculate the total amount of USDC needed in the operation above:

```
actual USDC amount needed = borrowed amount + makerFee + takerFee
actual USDC amount needed = 990e6 + 10e6 + 20e6
actual USDC amount needed = 1020e6
```

in the scenario above, the contract only held 1000e6 USDC but the operation require 1020e6 USDC. this would cause underflow.

in general scenario where there are multiple order, the USDC operated would exceed the matched amount and it would try to take USDC from other's limit order.

the same issue could also happen in the BORROW block.

Recommendation

the mitigation would be:

1. if the matched amount is 1000e6, borrowed amount accounted should be still 1000e6, but the ACTUAL amount that is sent to borrower address should be amount that deducted by the fee (eg: 990e6).
2. for lender side, the bond token minted should be (1000e6 * rate - fees). where the fees should be in form of bond token that later can be redeemed 1:1 to USDC by admin/fee receiver.

Low

[L-01] `CentuariCLOBMarketManager::deactivateMarket` is missing check if the current market still have user funds

Severity

Impact: Medium

Likelihood: Low

Description

`deactivateMarket` does not have check if the market still hold user funds.

also it would be difficult for an admin to deactivate market if they wait for user to withdraw their funds to avoid stuck fund inside the deactivated market.

Recommendation

if its needed to deactivate a market, add function to claim funds from user that affected by said market. so user can pull their fund from the deactivated market anytime.

[L-02] `CentuariCLOBMarketManager::deactivateMarket` always deactivate all maturity of given market config

Severity

Impact: Medium

Likelihood: Low

Description

Not all market with varying maturity would have same performance.

The current `deactivateMarket` implementation always deactivate all maturity in the given config.

Also there are case where market cannot be deactivated:

1. market config that is created by using `createSingleMarket` if it have unique maturity.
2. if beforehand the `setMaturities` is called to change the sets of maturity.

Recommendation

add parameter to select maturity of the market config that would be deactivated

[L-03] Inconsistent use of denominator when calculating fees

Severity

Impact: Low

Likelihood: Low

Description

When calculating fee, the denominator used is `100e18`. But there are multiple instance where the denominator used is `100e16` like in share calculation.

Recommendation

To avoid confusion and magic numbers, use BPS (1 BPS = 0.01%) and constant across the protocol for denominator.

```
uint256 public constant BPS_DENOMINATOR = 10_000;
```

so if we need percent calculation, we can just call `BPS_DENOMINATOR` as denominator.

[L-04] Invalid denominator used when creating LLTV metadata on CBT

Severity

Impact: Low

Likelihood: Low

Description

in the current implementation of CentuariBondToken.sol#L64 and CentuariBondToken.sol#L78, LLTV is divided by 1e14 when creating metadata of Bond Token.

This is incorrect because LLTV denoted in 16 decimals like 80e16.

Dividing using 1e14 would cause invalid LLTV shown in the bond token metadata.

Recommendation

use 100e16 instead of 1e14.

if possible, use BPS and constant when storing variable to avoid confusion.