



# **Centuari Audit Report**

Version 1.0

*KasturiSec*

July 9, 2025

# Centuari Audit Report

KasturiSec

July 8, 2025

Prepared by: KasturiSec

Auditors:

- JuggerNaut63
- 0xDemon
- farismaulana

## Table of Contents

- Table of Contents
- About
- Disclaimer
- About Centuari
- Risk Classification
- Audit Details
- Executive Summary
- Findings
- High
  - [H-01] `Centuari::addAnotherCollateral` does not transfer the collateral added from borrower
  - [H-02] Unhealthy Positions Cannot Be Liquidated Before Maturity
  - [H-03] `CENTUARI::withdrawCollateral` can be used to withdraw all collateral even if borrower still have debt
  - [H-04] Logic flaw in the `findMatchOrder()` function when the `oppositeOrderGroup` has `amount == 0` or status is `CANCELED`

- [H-05] Lender's order that is not matched cant be cancelled
- Medium
  - [M-01] No fee receiver set
  - [M-02] Users Can Liquidate Themselves Without Penalty
  - [M-03] Rate calculation disregarding the maturity days
  - [M-04] `Centuari::supplyCollateralAndBorrow` borrower health check only account amount without interest
  - [M-05] Partially Filled Market Orders Incorrectly Added to Queue
  - [M-06] if `placeMarketOrder` result in no match, would cause user fund stuck
  - [M-07] Incorrect fee handling when match order happening
- Low
  - [L-01] `CentuariCLOBMarketManager::deactivateMarket` is missing check if the current market still have user funds
  - [L-02] `CentuariCLOBMarketManager::deactivateMarket` always deactivate all maturity of given market config
  - [L-03] Inconsistent use of denominator when calculating fees
  - [L-04] Invalid denominator used when creating LLTV metadata on CBT

## 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
<b>Likelihood: High</b>	High	High	Medium
<b>Likelihood: Medium</b>	High	Medium	Low
<b>Likelihood: Low</b>	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

```
1 src/core/CentuariBondToken.sol
2 src/core/centuari-clob/CentuariCLOBMarketManager.sol
3 src/core/centuari-clob/CentuariCLOBMarket.sol
4 src/core/centuari-clob/CentuariCLOBOrderGroupManager.sol
5 src/core/centuari-clob/CentuariCLOB.sol
6 src/core/Centuari.sol
7 src/core/DataStore.sol
8 src/interfaces/centuari-clob/ICentuariCLOBMarket.sol
9 src/interfaces/centuari-clob/ICentuariCLOB.sol
10 src/interfaces/ICentuariCallbacks.sol
11 src/interfaces/ICentuari.sol
12 src/interfaces/IDataStore.sol
13 src/interfaces/IOracleManager.sol
14 src/libraries/centuari/CentuariDSLlib.sol
15 src/libraries/centuari/CentuariErrorsLib.sol
16 src/libraries/centuari/CentuariEventsLib.sol
17 src/libraries/centuari-clob/CentuariCLOBDSLlib.sol
18 src/libraries/centuari-clob/CentuariCLOBErrorsLib.sol
19 src/libraries/centuari-clob/CentuariCLOBEventsLib.sol
20 src/libraries/centuari-clob/CentuariCLOBOrderProcessorLib.sol
21 src/libraries/centuari-clob/CentuariCLOBValidator.sol
22 src/libraries/centuari-clob/OrderQueueLib.sol
23 src/libraries/MarketConfigLib.sol
24 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]	<code>Centuari::addAnotherCollateral</code> does not transfer the collateral added from borrower	High	Resolved
[H-02]	Unhealthy Positions Cannot Be Liquidated Before Maturity	High	Resolved
[H-03]	<code>CENTUARI::withdrawCollateral</code> can be used to withdraw all collateral even if borrower still have debt	High	Resolved
[H-04]	Logic flaw in the <code>findMatchOrder()</code> function when the <code>oppositeOrderGroup</code> has <code>amount == 0</code> or status is <code>CANCELED</code>	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]	<code>Centuari::supplyCollateralAndBorrow</code> 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 <code>placeMarketOrder</code> result in no match, would cause user fund stuck	Medium	Resolved
[M-07]	Incorrect fee handling when match order happening	Medium	Resolved
[L-01]	<code>CentuariCLOBMarketManager::deactivateMarket</code> is missing check if the current market still have user funds	Low	Resolved

ID	Title	Severity	Status
[L-02]	<code>CentuariCLOBMarketManager::deactivateMarket</code> 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`.

```
1 function addAnotherCollateral(MarketConfig memory config, uint256
  maturity, address user, uint256 amount)
2     external
3     nonReentrant
4     onlyActiveMarket(config.marketId())
5     onlyActiveMaturity(config.marketId(), maturity)
6     onlyActiveUser(config.marketId(), maturity, user)
```

```
7     whenNotPaused
8     {
9         if (amount == 0) revert CentuariErrorsLib.InvalidAmount();
10
11         IDataStore dataStore = IDataStore(dataStores[config.marketId()]);
12
13         // ✖ MAIN PROBLEM: Only updating storage without transferring token
14         CentuariDSLlib.setUserCollateral(dataStore, maturity, user,
15             CentuariDSLlib.getUserCollateral(dataStore, maturity, user) +
16             amount
17         );
18         // ✖ NO TOKEN TRANSFER FROM USER TO CONTRACT!
19         // MISSING: IERC20(collateralToken).safeTransferFrom(msg.sender,
20             address(this), amount);
21
22         emit CentuariEventsLib.AddAnotherCollateral(config.marketMaturityId(
23             maturity), user, amount);
24     }
```

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.

```
1 function _isHealthy(Id id, uint256 maturity, address user, uint256
2     collateral, uint256 amount) public view returns (bool) {
3     // Using data from storage that may contain phantom collateral
4     uint256 collateralValue = ((CentuariDSLlib.getUserCollateral(
5         dataStore, maturity, user) + collateral) * collateralPrice) /
6     collateralDecimals;
7     // Health check becomes inaccurate due to phantom collateral
8 }
```

## Recommendation

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

```
1 function addAnotherCollateral(MarketConfig memory config, uint256
2     maturity, address user, uint256 amount)
3     external
4     nonReentrant
5     onlyActiveMarket(config.marketId())
6     onlyActiveMaturity(config.marketId(), maturity)
7     onlyActiveUser(config.marketId(), maturity, user)
```



```
7     whenNotPaused
8     {
9         // Validate that amount cannot be zero
10        if (amount == 0) revert CentuariErrorsLib.InvalidAmount();
11
12        // Validation that only the user himself can add collateral
13    +    if (user != msg.sender) revert CentuariErrorsLib.InvalidUser();
14
15        // Get DataStore for the relevant market
16        IDataStore dataStore = IDataStore(dataStores[config.marketId()]);
17
18        // Get collateral token address
19    +    address collateralToken = dataStore.getAddress(CentuariDSLlib.
20        COLLATERAL_TOKEN_ADDRESS);
21
22        // ❌ FIX: Transfer tokens from user to contract first
23    +    IERC20(collateralToken).safeTransferFrom(msg.sender, address(this),
24        amount);
25
26        // Adding new collateral to existing collateral
27        CentuariDSLlib.setUserCollateral(dataStore, maturity, user,
28            CentuariDSLlib.getUserCollateral(dataStore, maturity, user) +
29            amount
30        );
31
32        // Emit event to record additional collateral
33        emit CentuariEventsLib.AddAnotherCollateral(config.marketMaturityId
34            (maturity), user, amount);
35    }
```

## [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.

```

1 function liquidate(MarketConfig memory config, uint256 maturity,
2   address user) external {
3   // ... other validations ...
4   // ❌ ISSUE: Use of the || operator wrong
5   if ((block.timestamp < maturity) || _isHealthy(config.marketId(),
6     maturity, user, 0, 0)) {
7     revert CentuariErrorsLib.LiquidationNotAllowed();
8   }
9   // ... liquidation logic ...
10 }

```

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)

```

1 Not Maturity, Healthy true || true = true REJECTED ❌TRUE
2 Not Maturity, Unhealthy true || false = true REJECTED ❌WRONG
3 Already Maturity, Healthy false || true = true REJECTED ❌WRONG
4 Already Maturity, Unhealthy false || false = false ALLOWED ❌TRUE

```

Lines 2 and 3 give incorrect results.

`&&` (AND):

```

1 Not Maturity, Healthy true && true = true REJECTED ❌TRUE
2 Not Maturity, Unhealthy true && false = false ALLOWED ❌TRUE
3 Already Maturity, Healthy false && true = false ALLOWED ❌NEED POLICY
4 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`.

```
1 function
  test_LiquidationLogicError_UnhealthyPositionBeforeMaturityCannotBeLiquidated
  () public {
2   // === SETUP BORROWING POSITION ===
3
4   // Setup borrower with collateral and borrow
5   address testBorrower = makeAddr("testBorrower");
6   _mintTokensAndSetApprovals(testBorrower);
7
8   // Test configuration
9   uint256 collateralAmount = 1e18; // 1 ETH
10  uint256 borrowAmount = 1800e6; // 1800 USDC
11  uint256 borrowRate = 5e16; // 5% rate
12
13  // Initial ETH price: $2500 (from BaseTest setup)
14  // Initial health ratio: (1 ETH * $2500) / $1800 = 138% > 80% (
    HEALTHY)
15
16  // Borrower performs supply collateral and borrow
17  vm.prank(address(centuariCLOB));
18  bool borrowSuccess = centuari.supplyCollateralAndBorrow(
19    lendMarketConfigs[0],
20    testMaturity,
21    testBorrower,
22    borrowRate,
23    borrowAmount,
24    collateralAmount
25  );
26  assertTrue(borrowSuccess, "Initial borrow should succeed");
27
28  // Verify initial position is healthy
29  bool initialHealthy = centuari._isHealthy(
30    lendMarketConfigs[0].marketId(),
31    testMaturity,
32    testBorrower,
33    0,
34    0
35  );
36  assertTrue(initialHealthy, "Position should be initially healthy");
37
38  // === PRICE MANIPULATION TO MAKE POSITION UNHEALTHY ===
39
40  // Lower ETH price from $2500 to $1400
41  // New health ratio: (1 ETH * $1400) / $1800 = 77% < 80% (UNHEALTHY
    !)
42
43  // Get oracle for ETH/USDC
44  address ethUsdcOracle = oracleManager.getOracle(
45    address(mockTokens[0]), // WETH
46    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
65     // days)
66     assertTrue(block.timestamp < testMaturity, "Should be before
67     maturity");
68
69     // === TRY LIQUIDATION (SHOULD SUCCEED BUT WILL FAIL DUE TO BUG)
70     ===
71
72     // Setup liquidator
73     address testLiquidator = makeAddr("testLiquidator");
74     _mintTokensAndSetApprovals(testLiquidator);
75
76     // Try liquidation - this should SUCCEED because position is
77     // unhealthy
78     // But will FAIL due to bug in liquidation logic
79     vm.prank(testLiquidator);
80     vm.expectRevert(CentuariErrorsLib.LiquidationNotAllowed.selector);
81     centuari.liquidate(lendMarketConfigs[0], testMaturity, testBorrower
82     );
83
84     // === PROOF OF BUG: UNHEALTHY POSITION CANNOT BE LIQUIDATED ===
85
86     console.log("=== PROOF OF LIQUIDATION LOGIC ===");
87     console.log("Condition: Before maturity =", block.timestamp <
88     testMaturity);
89     console.log("Condition: Position healthy =", !nowUnhealthy);
90
91     // === SIMULATION IF PRICE CONTINUES TO DROP ===
92
93     // Lower price again to $1000
94     MockOracle(ethUsdcOracle).setPrice(1000e6);
95
96     // Health ratio now: $1000 / $1800 = 55% (VERY UNHEALTHY!)
97     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         );
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,
2      address user)
3      external
4      nonReentrant
   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(),
11 maturity, user, 0, 0)) {
12 // ✖ FIX: Liquidation is only rejected if BOTH conditions are true
13 // 1. Not yet mature AND 2. Position is still healthy
14 // This allows liquidation of unhealthy positions before maturity
15 +    if ((block.timestamp < maturity) && !_isHealthy(config.marketId(),
16 maturity, user, 0, 0)) {
17         revert CentuariErrorsLib.LiquidationNotAllowed();
18     }
19     // ... other code ...
20 }
```

**[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 `withdrawCollateral` in Centuari.sol#L318-L319:

```
1     function withdrawCollateral(MarketConfig memory config, uint256
2         maturity, uint256 amount)
3     {
4     ...
5         uint256 userCollateral = CentuariDSLlib.getUserCollateral(
6             datastore, maturity, msg.sender);
7         if (userCollateral < amount) revert CentuariErrorsLib.
8             InsufficientCollateral();
9
10        CentuariDSLlib.setUserCollateral(dataStore, maturity, msg.
11            sender, userCollateral - amount);
12        if (!_isHealthy(config.marketId(), maturity, msg.sender,
13            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`:

```
1      function _isHealthy(Id id, uint256 maturity, address user, uint256
2          collateral, uint256 amount) public view returns (bool) {
3          IDataStore dataStore = IDataStore(dataStores[id]);
4
5          address oracle = ORACLE_MANAGER.getOracle(dataStore.getAddress(
6              CentuariDSLlib.COLLATERAL_TOKEN_ADDRESS), dataStore.
7              getAddress(CentuariDSLlib.LOAN_TOKEN_ADDRESS));
8          uint256 collateralPrice = IMockOracle(oracle).price();
9          uint256 collateralDecimals = 10 ** IERC20Metadata(dataStore.
10              getAddress(CentuariDSLlib.COLLATERAL_TOKEN_ADDRESS)).decimals
11              ();
12
13          // borrow 1000USDC in 5% in one year, borrowed value will be
14          // 1050USDC
15          uint256 borrowedValue = CentuariDSLlib.getUserBorrowValue(
16              dataStore, maturity, user) + amount;
17 @>      uint256 collateralValue = ((CentuariDSLlib.getUserCollateral(
18              dataStore, maturity, user) + collateral) * collateralPrice) /
19              collateralDecimals;
20          uint256 maxBorrowedValue = (collateralValue * dataStore.getUint
21              (CentuariDSLlib.LLTV_UINT256)) / 100e16;
22
23          return borrowedValue <= maxBorrowedValue;
24      }
```

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`:

```
1      CentuariDSLlib.setUserCollateral(dataStore, maturity, msg.sender
2          , userCollateral - amount);
3 -      if (!_isHealthy(config.marketId(), maturity, msg.sender, amount
4          , 0)) revert CentuariErrorsLib.InsufficientCollateral();
5 +      if (!_isHealthy(config.marketId(), maturity, msg.sender, 0, 0))
6          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



```
1  if(oppositeOrderGroupAmount == 0 || oppositeOrderGroupStatus ==  
    OrderStatus.CANCELLED) {  
2  -      oppositeOrderId = nextoppositeOrderId;  
3      removeOrderFromQueue(context, marketId, rate, oppositeSide,  
        oppositeOrderId);  
4  +      oppositeOrderId = nextoppositeOrderId;  
5      continue;  
6  }
```

## [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(),
13        maturity, user, 0, 0)) {
14        revert CentuariErrorsLib.LiquidationNotAllowed();
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,
2         address user, uint256 rate, uint256 amount)
3     ...
4     // mint tokenized bond to the lender
5     @> 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,
2           amount)) return false;
3           IDataStore dataStore = IDataStore(dataStores[config.marketId()
4           ]);
5           CentuariDSLlib.setUserCollateral(dataStore, maturity, user,
6           CentuariDSLlib.getUserCollateral(dataStore, maturity, user)
7           + collateral
8           );
9  @>      uint256 userBorrowValue = CentuariDSLlib.getUserBorrowValue(
10         dataStore, maturity, user);
11 @>      uint256 additionalBorrowValue = amount + ((amount * rate) / 1
12         e18);
13 @>      CentuariDSLlib.setUserBorrowValue(dataStore, maturity, user,
14         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 ltv 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     {
3         // Market Orders are added to the queue if they are partially
4         // filled - THIS IS WRONG!
5         CentuariCLOBOrderProcessorLib.addOrderToQueue(address(market),
6             marketId, msg.sender, matchRate, side, orderId, groupId);
7     }
```

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:

```
1 actual USDC amount needed = borrowed amount + makerFee + takerFee
2 actual USDC amount needed = 990e6 + 10e6 + 20e6
3 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  $(1000e6 * \text{rate} - \text{fees})$ . 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.