Pashov Audit Group

Avon Security Review

July 15th 2025 - August 4th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Risk Classification

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

4. About Avon

Avon is a modular DeFi lending protocol that combines isolated ERC4626 lending pools, a vault-based liquidity manager, and an orderbook system to enable capital-efficient borrowing and optimized yield distribution. It offers flexible collateral management, dynamic risk parameters, and automated interest rate models while ensuring isolated risk per pool through customizable configurations.

5. Executive Summary

A time-boxed security review of the **avon-xyz/avon-core** and **avon-xyz/avon-periphery** repositories was done by Pashov Audit Group, during which Pashov Audit Group engaged to review **Avon**. A total of **60** issues were uncovered.

Protocol Summary	
Project Name	Avon
Protocol Type	Lending protocol
Timeline	July 15th 2025 - August 4th 2025

Review commit hashes:

- 09da16173e2c0e81ae2279e429efac00a877f2f2 (avon-xyz/avon-core)
- 4611308f08ea32aed9b931a9abdc89b0975987d6 (avon-xyz/avon-periphery)

Fixes review commit hashes:

- 028f3a969efa63438a0d2523e91326df12c1e52b (avon-xyz/avon-core)
- 97b7a0e71e41c7094b2f9a4d8fdad87a506bfb51 (avon-xyz/avon-periphery)

Scope

`Orderbook.sol` `OrderbookFactory.sol` `OrderbookFactoryStorage.sol` `ErrorsLib.sol` `EventsLib.sol` `MathLib.sol` `OrderbookLib.sol` `RedBlackTreeLib.sol` `AvonPoolFactory.sol` `VaultFactory.sol` `LiquidityAllocator.sol` `SharesLib.sol` `AccrueInterest.sol` `BorrowRepay.sol` `CollateralManagement.sol` `DepositWithdraw.sol` `FlashLoan.sol` `Liquidation.sol` `PositionGuard.sol` `UpdateOrders.sol` `PoolConstants.sol` `PoolErrors.sol` `PoolEvents.sol` `PoolGetter.sol` `AvonPool.sol` `PoolStorage.sol` `Vault.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
Critical	2
High	5
Medium	15
Low	38
Total findings	**60**

Summary of findings

ID	Title	Severity	Status
[C-01]	Complete pool freeze due to `ABDKMath64x64` overflow in interest accrual	Critical	Resolved
[C-02]	Flash loan fees will remain permanently locked in the contract	Critical	Resolved
[H-01]	Fee share miscalculation may reduce protocol and manager revenue	High	Resolved
[H-02]	Interest accrual timestamp not updated, causing borrower overcharge	High	Resolved
[H-03]	Performance fee is incorrectly charged on user principal during deposits	High	Resolved
[H-04]	First borrower can DoS `AvonPool` by inflating `totalBorrowShares`	High	Resolved
[H-05]	The vault's `_accrueInterest` fee share calculation is incorrect	High	Resolved
[M-01]	Flawed liquidity checks may cause pool insolvency	Medium	Resolved
[M-02]	Orderbook management missing during pool pause/unpause	Medium	Resolved
[M-03]	Remove liquidity lacks pool liquidity validation	Medium	Resolved
[M-04]	Inconsistent seize cap between collateral and share-based liquidation	Medium	Resolved

ID	Title	Severity	Status
[M-05]	Interest accrual during pause blocks repayment, causing liquidations	Medium	Resolved
[M-06]	Incorrect fee share calculation in preview function uses stale supply assets	Medium	Resolved
[M-07]	Heap violation in partial fills breaks liquidity matching priority	Medium	Resolved
[M-08]	Heap invariant violation in `removeEntry` causes order mismatch	Medium	Resolved
[M-09]	`_repayWithExactShares` does not update state, miscalculating collateral	Medium	Resolved
[M-10]	Missing `accrueInterest()` call before updating manager and protocol fees	Medium	Resolved
[M-11]	`managerFee` and `protocolFeeAmount` bypassed with frequent `accrueInterest()`	Medium	Resolved
[M-12]	`removeQueueEntry` may incorrectly reset `depositHead`	Medium	Resolved
[M-13]	`MAX_LIMIT_ORDERS` check in `insertLimitBorrowOrder` may block valid updates	Medium	Resolved
[M-14]	Large pools fail to quote up to 99% of their available liquidity	Medium	Resolved
[M-15]	Lack of safety buffer between borrow and liquidation thresholds	Medium	Acknowledged
[L-01]	`previewBorrow` missing available liquidity validation	Low	Resolved
[L-02]	Existing whitelisted pools lack `IRM` validation after `IRM` disabling	Low	Acknowledged
[L-03]	APY calculation silently fails when result is zero	Low	Resolved
[L-04]	`RC4626` max functions ignore vault withdrawal constraints	Low	Acknowledged
[L-05]	Withdrawal queue remains uncleared when pool shares are depleted	Low	Acknowledged
[L-06]	Liquidation bonus and seize cap have inconsistent scaling	Low	Acknowledged

ID	Title	Severity	Status
[L-07]	Lack of validation in `borrow()` permits ambiguous parameters	Low	Resolved
[L-08]	Redundant logic in liquidity allocation from impossible conditions	Low	Resolved
[L-09]	Lack of `share == 0` check in `_deposit` function	Low	Resolved
[L-10]	No slippage control in deposit/withdraw leads to bad exchange rates	Low	Acknowledged
[L-11]	EIP-4626: max-functions overlook paused state	Low	Acknowledged
[L-12]	Pool tokens with varied fees share same name and symbol	Low	Acknowledged
[L-13]	Inconsistent share to asset conversion in position safety check	Low	Resolved
[L-14]	Interest overestimated in `_previewAccrueInterest()` due to quote buffer	Low	Acknowledged
[L-15]	Incorrect APY calculation from updated total supply assets	Low	Resolved
[L-16]	Advertised borrow rate can be misleadingly low	Low	Acknowledged
[L-17]	Lack of timelock in `increaseLLTV()` allows immediate risk changes	Low	Acknowledged
[L-18]	Inefficient order matching can lead to revert	Low	Acknowledged
[L-19]	Silent failure in order cancellation when there's no match	Low	Resolved
[L-20]	Missing `nonReentrant` modifier in `cancelBorrowOrder` function	Low	Resolved
[L-21]	Users cannot control collateral requirements in market borrow orders	Low	Resolved
[L-22]	`totalAssets` function can revert due to gas limit exceeded when managing many pools	Low	Acknowledged
[L-23]	Access bypass via `TimelockController` direct interaction	Low	Acknowledged

ID	Title	Severity	Status
[L-24]	`AvonPool`'s withdrawal and `borrow` operations are vulnerable to griefing	Low	Acknowledged
[L-25]	Missing maximum amount check borrowing with shares	Low	Acknowledged
[L-26]	Pausing repay may cause liquidation risk when pool is unpaused	Low	**Resolved**
[L-27]	`totalCount` may exceed counterParty length causing revert	Low	**Resolved**
[L-28]	Inconsistent interest calculations lead to inflated APY	Low	**Resolved**
[L-29]	Unbounded loop in `_cancelPoolOrders()` may cause out-of-gas	Low	Acknowledged
[L-30]	Fee checked only at proposal time lets total fee exceed max limit	Low	**Resolved**
[L-31]	Insufficient incentives for small liquidations risk bad debt	Low	Acknowledged
[L-32]	Incorrect slippage control in `borrow` function	Low	Acknowledged
[L-33]	Stale roles persist for former owners due to direct role assignment	Low	Acknowledged
[L-34]	Pool position loss in orderbook due to order reinsertion on liquidity changes	Low	Acknowledged
[L-35]	Out of gas issue in `_performWithdraw` function	Low	Acknowledged
[L-36]	In `cancelBorrowOrder()`, user is forced to cancel minimum loan amount	Low	Acknowledged
[L-37]	Vault's `_accrueInterest` updates `prevTotal` even when at a loss	Low	Acknowledged
[L-38]	Bad borrowers reclaim collateral on insolvency, lenders absorb loss	Low	Acknowledged

Critical findings

[C-01] Complete pool freeze due to `ABDKMath64x64` overflow in interest accrual

Severity

Impact: High

Likelihood: High

Description

The `_accrueInterest` function contains a critical overflow vulnerability that causes a complete pool freeze when calculating APY. The issue occurs when converting `totalSupplyAssets * elapsed` to ABDKMath64x64 fixed-point format:

```
    function _accrueInterest(PoolStorage.PoolState storage s) internal returns (uint256
managerFeeShares, uint256 protocolFeeShares) {
        ...
            int128 lenderRate = ABDKMath64x64.div(
                ABDKMath64x64.fromUInt(accruedInterestWithFees),
                ABDKMath64x64.fromUInt(totalSupplyAssets * elapsed) // @audit revert here
            );
            ...
        }
        ...
    }
```

The ABDKMath64x64 library's `fromUInt` function enforces a strict limit:

```
  function fromUInt (uint256 x) internal pure returns (int128) {
    unchecked {
      require (x <= 0x7FFFFFFFFFFFFFFF); // 2**63-1 ~ 9.2e18
      return int128 (int256 (x << 64));
    }
  }
```

So we have this threshold: - Maximum convertible value: `2^63 - 1 ≈ 9.22 × 10^18` . - For 18-decimal tokens: ~9.22 tokens maximum. - Overflow occurs when: `totalSupplyAssets * elapsed > 9.22 × 10^18` .

Once the overflow occurs, all core functions become permanently unusable because they call `accrueInterest` .

Therefore, if the loan tokens are 18 decimals, such as WETH or DAI, and the total supply of assets is more than 9.2e18. The `_accrueInterest` will always revert. All deposited funds become permanently locked.

Consider a scenario:

- User deposits 10 ETH into pool.
- A borrower borrows 0.1 ETH.
- 1 second later, all operation fails because `s._accrueInterest` reverts.
- 10 ETH is lost.

The same vulnerability exists in `_previewAccrueInterest`, affecting view functions and external integrations.

Recommendations

Remove the APY calculation from `_accrueInterest` function. It can be calculated off chain.

[C-02] Flash loan fees will remain permanently locked in the contract

Severity

Impact: High

Likelihood: High

Description

The protocol allows users to take flash loans using tokens deposited by lenders.

```
library FlashLoan {
    using Math for uint256;

    function _flashLoan(
        PoolStorage.PoolState storage s,
        address token,
        uint256 assets,
        bytes calldata data
    ) internal {
        if (assets == 0) revert PoolErrors.ZeroAddress();
        if (token != s.config.loanToken) revert PoolErrors.InvalidInput();
        if (s.totalBorrowAssets > s.totalSupplyAssets) revert
PoolErrors.InsufficientLiquidity();

        // Calculate flash loan fee
        uint256 feeAmount = assets.mulDiv(s.flashLoanFee, PoolConstants.WAD,
Math.Rounding.Ceil);

        emit PoolEvents.FlashLoan(msg.sender, token, assets);

        SafeERC20.safeTransfer(ERC20(token), msg.sender, assets);

        IAvonFlashLoanCallback(msg.sender).onAvonFlashLoan(assets, data);
```

```
        SafeERC20.safeTransferFrom(ERC20(token), msg.sender, address(this), assets + feeAmount);
    }
}
```

As you can see, a fee must be paid if `flashLoanFee != 0`. The issue is that these fees are sent to the **AvoonPool**, which does not implement any function to withdraw them. Additionally, `accrueInterest()` only accounts for interest accrued by lenders based on borrowed amounts (internal accounting), not on the actual token balance of the contract.

As a result, the contract's token balance will increase due to these fees, but they are neither reflected in the internal accounting nor withdrawable by the lenders.

To reproduce the issue, copy the following POC into `FlashLoanTest.t`. To run the test successfully in the mock contract's `onAvonFlashLoan()`, you need to set the allowance to `type(uint256).max`.

```
function testBlockedFlashLoanFees() public {
        uint256 loanAmount = 500e6;
        bytes memory data = abi.encode("flash loan data");

        // Setup the flash loan fee
        vm.prank(manager);
        pool.updateFlashLoanFee(0.01e18);

        vm.warp(block.timestamp + 1 weeks);

        vm.prank(manager);
        pool.execute(address(pool),
        0,
        abi.encodeWithSelector(
                pool._executeUpdateFlashLoanFee.selector,
                0.01e18
            ),
            bytes32(0),
            bytes32(keccak256("UPDATE_FLASH_LOAN_FEE"))
        );

        uint256 feeAmount = (loanAmount * pool.getFlashLoanFee()) / 1e18;

        // Mint fees to repay flashLoan fees
        vm.prank(owner);
        loanToken.mint(address(flashBorrower), feeAmount);

        // Execute flash loan
        vm.prank(address(flashBorrower));
        pool.flashLoan(address(loanToken), loanAmount, data);

        // Verify flash loan was processed correctly
        assertEq(flashBorrower.lastLoanAmount(), loanAmount, "Flash loan amount should match");
        assertEq(flashBorrower.lastData(), data, "Flash loan data should match");

        //borrow some amount to accrue interest
        uint256 borrowAmount = 100e6;
        uint256 collateralAmount = pool.previewBorrow(borrower, borrowAmount,
```

```
DEFAULT_COLLATERAL_BUFFER);

        vm.startPrank(borrower);

        // Deposit collateral
        collateralToken.approve(address(pool), collateralAmount);
        pool.depositCollateral(collateralAmount, borrower);

        (uint256 assets, uint256 shares) = pool.borrow(
            borrowAmount,
            0,
            borrower,
            borrower,
            borrowAmount
        );

        // Simulate time passing to accrue interest
        vm.warp(block.timestamp + 3 weeks);

        // Repay borrowed amount
        loanToken.approve(address(pool), assets);
        pool.repay(assets, 0, address(borrower));

        vm.stopPrank();

        vm.prank(lender1);
        pool.withdraw(DEFAULT_DEPOSIT_AMOUNT, address(lender1), address(lender1));

        assertEq(loanToken.balanceOf(address(pool)), feeAmount);
    }
```

As you can see, when the lender withdraws their deposited amount, the contract's remaining balance equals the `feeAmount`, representing the stuck tokens.

In this case, since there is only one lender, the `feeAmount` should either be received by that lender, or, if it is intended to be earned by the protocol team, a function to withdraw it is missing.

In addition, the flash loan function changes the pool's `liquidity` but doesn't update the `orderbook`:

```
function flashLoan(
    address token,
    uint256 assets,
    bytes calldata data
) external whenNotPaused {
    PoolStorage._state()._flashLoan(token, assets, data);
    // Missing: _updateOrders() call
}
```

The net change is the `feeAmount` of the `flashloan` (positive for the pool). Thus, the available liquidity changes. However, the orderbook still reflects old liquidity levels.

Recommendations

If these fees are meant to be accounted for by the lenders, they should be integrated into `accruedInterest()`. However, if the fees are intended for the protocol, a withdrawal function must be implemented to claim them. In addition, add `_updateOrders` call.

High findings

[H-01] Fee share miscalculation may reduce protocol and manager revenue

Severity

Impact: Medium

Likelihood: High

Description

When interest is accrued, a portion is allocated to the manager and the protocol as fees. This is done by minting new pool shares and assigning them to the respective fee recipients. The conversion from the fee amount (in assets) to shares should be based on the current share price.

Below is the `_accrueInterest` function:

```
        // Calculate manager fee shares if applicable
        if (managerFee != 0 && accruedInterest > 0) {
            managerFeeAmount = accruedInterest.mulDiv(managerFee, PoolConstants.WAD);
            managerFeeShares = managerFeeAmount.mulDiv(totalSupplyShares, totalSupplyAssets -
managerFeeAmount);
            totalNewShares += managerFeeShares;
        }

        // Calculate protocol fee shares if there's interest
        if (accruedInterest > 0) {
            protocolFeeAmount =
accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(), PoolConstants.WAD);
            protocolFeeShares = protocolFeeAmount.mulDiv(totalSupplyShares, totalSupplyAssets -
protocolFeeAmount);
            totalNewShares += protocolFeeShares;
        }
```

The correct share price is determined by the total assets belonging to existing shareholders divided by the total number of existing shares. After interest accrues, the total assets backing the original `totalSupplyShares` is `(totalSupplyAssets + accruedInterest) - managerFeeAmount - protocolFeeAmount`.

However, the current implementation calculates the shares for each fee type using a denominator that only subtracts its own fee amount, not all fees.

To be direct, the denominator `totalSupplyAssets - managerFeeAmount` is incorrect because it doesn't account for the `protocolFeeAmount` that is also being taken from the accrued interest. The denominator `totalSupplyAssets - protocolFeeAmount` is incorrect because it doesn't account for the `managerFeeAmount`.

Because both denominators are larger than they should be, the number of minted shares for both the manager and the protocol is consistently lower than the amount they are owed.

Recommendations

To fix this issue, a single, correct denominator should be calculated first and then used for both fee share calculations.

```
// ...

if (accruedInterest > 0) {
    // First, calculate both fee amounts
    managerFeeAmount = accruedInterest.mulDiv(managerFee, PoolConstants.WAD);
    protocolFeeAmount =
accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(), PoolConstants.WAD);

    // Calculate the correct total assets backing the original shares
    uint256 assetsBackingShares = totalSupplyAssets - managerFeeAmount - protocolFeeAmount;

    // Calculate manager fee shares if applicable
    if (managerFeeAmount > 0) {
        managerFeeShares = managerFeeAmount.mulDiv(totalSupplyShares, assetsBackingShares);
        totalNewShares += managerFeeShares;
    }

    // Calculate protocol fee shares
    if (protocolFeeAmount > 0) {
        protocolFeeShares = protocolFeeAmount.mulDiv(totalSupplyShares, assetsBackingShares);
        totalNewShares += protocolFeeShares;
    }
}
```

[H-02] Interest accrual timestamp not updated, causing borrower overcharge

Severity

Impact: Medium

Likelihood: High

Description

The `_accrueInterest` has a flaw where it returns early when `accruedInterest == 0` without updating the `s.lastUpdate` timestamp. This leads to borrowers being charged interest for periods when no assets were actually borrowed.

```
    function _accrueInterest(PoolStorage.PoolState storage s) internal returns (uint256
managerFeeShares, uint256 protocolFeeShares) {
        ...
        uint256 accruedInterest = totalBorrowAssets.mulDiv(expFactor - PoolConstants.WAD,
PoolConstants.WAD);
        if (accruedInterest == 0) return (0, 0); // Early return without updating lastUpdate
        ...
    }
```

Consider a scenario:

- T0: Pool is created, lastUpdate = T0, totalBorrowAssets = 0.
- T1: Someone deposits funds, triggering `_accrueInterest`:
  - Since `totalBorrowAssets = 0`, `accruedInterest = 0`.
  - Function returns early, lastUpdate remains T0.
- T2: First borrower takes a loan of 1000 tokens:
  - `_accrueInterest` is called first, but still sees `totalBorrowAssets = 0`, so `accruedInterest = 0`.
  - Function returns early again, `lastUpdate` still remains T0.
  - Then `_borrow` updates `totalBorrowAssets = 1000`.
- T3: Next operation triggers _accrueInterest():
  - elapsed = T3 - T0 (the entire time since pool creation!).
  - Interest calculated on 1000 tokens for the period (T3 - T0).
  - Borrower is overcharged for the period T0 to T2 when they had no loan.

This means the borrower pays interest as if they had borrowed the funds from pool creation time T0, rather than from their actual borrow time T2. The first borrowers are systematically overcharged interest for periods before their loans existed for every pool.

Recommendations

Update `s.lastUpdate` even when no interest is accrued.

[H-03] Performance fee is incorrectly charged on user principal during deposits

Severity

Impact: Medium

Likelihood: High

Description

A flaw in the Vault contract's fee accrual mechanism causes manager performance fees to be incorrectly charged on user deposits, treating fresh principal as "gains" that warrant a performance fee. This results in immediate value extraction from depositors before their capital has any opportunity to generate yield.

The critical flaw lies in the order of operations: 1. A user calls `deposit(amount, ...)`. 2. `_accrueInterest()` executes first. It calculates any real yield, takes a fee on it, and then sets `prevTotal = totalAssets()`. At this point, the new deposit has not yet arrived. 3. `_accrueInterest()` finishes. 4. The `super.deposit()` logic runs, and the user's `amount` is transferred into the vault. The vault's `totalAssets()` value is now higher by `amount`. 5. `prevTotal`, however, remains at the pre-deposit value. 6. The **next time** any user action triggers `_accrueInterest()`, the function sees `currentAssets` (which includes the previous deposit) as being greater than `prevTotal`. It incorrectly identifies the previous user's principal deposit as a "gain" and charges a performance fee on it.

This effectively means the manager is paid a fee on fresh capital that has had no time to generate any yield.

```
function deposit(uint256 assets, address receiver) public override nonReentrant returns
(uint256 shares) {
    _accrueInterest();                          // Sets prevTotal = totalAssets() (without new
deposit)
    shares = super.deposit(assets, receiver);   // Increases totalAssets() by deposit amount
    _allocateDeposit(assets);
}

function _accrueInterest() internal {
    uint256 currentAssets = totalAssets();

    if (prevTotal > 0 && currentAssets > prevTotal) {
        uint256 gain = currentAssets -
prevTotal;   // Incorrectly includes previous deposits as "gain"
        uint256 managerFeesAmount = gain.mulDiv(managerFees, 1e18);

        if (managerFeesAmount > 0) {
            uint256 shares = convertToShares(managerFeesAmount);
            _mint(feeRecipient, shares);
        }
    }
```

```
    prevTotal = currentAssets;  // Updated before accounting for the incoming deposit
}
```

Consider a scenario:

- Initial State: Vault holds 1000 USDC, prevTotal = 1000.
- Alice deposits 500 USDC:
  - `_accrueInterest` executes: currentAssets = 1000, prevTotal = 1000, no fee charged.
  - `prevTotal` updated to 1000.
  - `super.deposit` executes: vault now holds 1500 USDC.
- Bob deposits 100 USDC (or any user action triggering _accrueInterest):
  - `_accrueInterest` executes: currentAssets = 1500, prevTotal = 1000.
  - Calculated "gain" = 500 USDC (Alice's principal).
  - Manager fee charged on 500 USDC, diluting Alice's shares.
- Result: Alice immediately loses a percentage of her principal to performance fees.

The impact is: - every depositor loses a portion of their principal to fees instantly. - fees are paid through share minting, diluting all existing shareholders.

Recommendations

Modify the `deposit`, `mint`, `withdraw`, and `redeem` functions to correctly adjust the `prevTotal` benchmark *after* the main token transfer has occurred.

[H-04] First borrower can DoS `AvonPool` by inflating `totalBorrowShares`

Severity

Impact: Medium

Likelihood: High

Description

When a `borrow` is requested and the number of `shares` is provided, the corresponding `assets` are calculated using `toAssetsDown`. Due to the rounding down behavior, users can receive `shares` without actually receiving any borrowed `assets`.

```
    function _borrowWithExactShares(
        PoolStorage.PoolState storage s,
        uint256 shares,
```

```
        address onBehalf,
        address receiver,
        uint256 minAmountExpected
    ) internal returns (uint256, uint256) {
        if (shares == 0) revert PoolErrors.InvalidInput();
        if (receiver == address(0)) revert PoolErrors.ZeroAddress();
        if (!s._isSenderPermitted(onBehalf)) revert PoolErrors.Unauthorized();

>>>     uint256 assets = shares.toAssetsDown(s.totalBorrowAssets, s.totalBorrowShares);

        s.positions[onBehalf].borrowShares += shares;
        s.totalBorrowShares += shares;
        s.totalBorrowAssets += assets;
        s.positions[onBehalf].poolBorrowAssets = s.totalBorrowAssets;
        s.positions[onBehalf].poolBorrowShares = s.totalBorrowShares;
        s.positions[onBehalf].updatedAt = s.lastUpdate;

        if (!s._isPositionSafe(onBehalf)) revert PoolErrors.InsufficientCollateral();
        if (s.totalBorrowAssets > s.totalSupplyAssets) revert
PoolErrors.InsufficientLiquidity();
        if (assets < minAmountExpected) revert PoolErrors.InsufficientAmountReceived();

        SafeERC20.safeTransfer(ERC20(s.config.loanToken), receiver, assets);

        emit PoolEvents.Borrow(
            address(this),
            msg.sender,
            onBehalf,
            receiver,
            assets,
            shares
        );

        return (assets, shares);
    }
```

This can be abused: the first borrower can repeatedly request `borrow` to inflate `totalBorrowShares` until subsequent `borrow` calls revert due to overflow.

```
    function testGriefBorrow() public {
        // uint256 borrowAmount = 500e6;
        uint256 collateralAmount = 1e18;

        vm.startPrank(borrower);

        // Deposit collateral
        collateralToken.approve(address(pool), collateralAmount);
        pool.depositCollateral(collateralAmount, borrower);
        uint256 totalBorrowShares = 1e6;

        for (uint i = 0; i < 225; i++) {
            (uint256 assets, uint256 shares) = pool.borrow(
                0,
                totalBorrowShares - 1,
                borrower,
                borrower,
                0 // Minimum expected
```

```
            );
            totalBorrowShares += shares;
        }

        vm.expectRevert();
        (uint256 assets, uint256 shares) = pool.borrow(
                1e6,
                0,
                borrower,
                borrower,
                1 // Minimum expected
        );
        vm.stopPrank();
    }
```

Recommendations

Consider adding a minimum borrow amount for the first borrower.

[H-05] The vault's `_accrueInterest` fee share calculation is incorrect

Severity

Impact: Medium

Likelihood: High

Description

When the vault's `_accrueInterest` is called, it calculates `managerFeesAmount` based on the accrued `gain`, then uses `convertToShares` to determine the corresponding shares.

```
    function _accrueInterest() internal {
        uint256 currentAssets = totalAssets();

        // Only calculate fees if this isn't the first accrual and there's a gain
        if (prevTotal > 0 && currentAssets > prevTotal) {
            // Calculate the gain (interest earned)
            uint256 gain = currentAssets - prevTotal;

            // Calculate manager's share of the gains
            uint256 managerFeesAmount = gain.mulDiv(managerFees, 1e18);

            if (managerFeesAmount > 0) {
>>>             uint256 shares = convertToShares(managerFeesAmount);
                if (shares > 0) {
                    _mint(feeRecipient, shares);
                    emit ManagerFeesAccrued(managerFeesAmount, shares);
                }
            }
        }
```

```
        // Update the previous total assets amount for next accrual
        prevTotal = currentAssets;
        emit TotalAssetsUpdated(currentAssets);
    }
```

This results in an incorrect number of shares, as `convertToShares` includes `managerFeesAmount` in `totalAssets`. This will cause the minted shares to be worth less than the calculated `managerFeesAmount`.

Recommendations

```
// ...
            if (managerFeesAmount > 0) {
-               uint256 shares = convertToShares(managerFeesAmount);
+               uint256 shares = managerFeesAmount.mulDivDown(totalSupply(), totalAssets() -
managerFeesAmount);
                if (shares > 0) {
                    _mint(feeRecipient, shares);
                    emit ManagerFeesAccrued(managerFeesAmount, shares);
                }
            }
// ...
```

Medium findings

[M-01] Flawed liquidity checks may cause pool insolvency

Severity

Impact: Medium

Likelihood: Medium

Description

The `DepositWithdraw._withdraw` and `_redeem` functions perform a solvency check *before* updating the pool's state. The check ensures that `s.totalBorrowAssets` is not greater than `s.totalSupplyAssets` at the beginning of the operation. However, the `s.totalSupplyAssets` is only reduced after the check has passed and after the underlying `ERC4626.withdraw` call has already succeeded.

```
    function _withdraw(
        PoolStorage.PoolState storage s,
        uint256 assets,
        uint256 shares,
        address receiver
    ) internal {
        ...
        if (s.totalBorrowAssets > s.totalSupplyAssets) revert
PoolErrors.InsufficientLiquidity();

        s.totalSupplyAssets -= assets;  // @audit State updated after check
        s.totalSupplyShares -= shares;

        ...
    }
```

If the contract holds "extra" tokens not accounted for in `s.totalSupplyAssets` (e.g., from flash loan fees), an attacker can withdraw these tokens. The check will pass because the contract's ERC20 balance is sufficient and the subsequent state update leaves the pool insolvent, with `s.totalBorrowAssets > s.totalSupplyAssets`.

Currently, this issue is accidentally prevented because the calling function, `AvonPool.withdraw` and `_redeem`, subsequently calls `_updateOrders`, which would revert due to an underflow on the line `s.totalSupplyAssets - s.totalBorrowAssets`. However, relying on this accidental revert in a downstream function is a fragile and unreliable safety mechanism.

Recommendation

Apply the following fix to `DepositWithdraw._withdraw` and the equivalent logic in `DepositWithdraw._redeem`.

```
function _withdraw(
    PoolStorage.PoolState storage s,
    uint256 assets,
    uint256 shares,
    address receiver
) internal {
    if (assets == 0) revert PoolErrors.ZeroAssets();
    if (receiver == address(0)) revert PoolErrors.ZeroAddress();

+   uint256 newSupplyAssets = s.totalSupplyAssets - assets;
-   if (s.totalBorrowAssets > s.totalSupplyAssets) revert PoolErrors.InsufficientLiquidity();
+   if (s.totalBorrowAssets > newSupplyAssets) revert PoolErrors.InsufficientLiquidity();

+   s.totalSupplyAssets = newSupplyAssets;
-   s.totalSupplyAssets -= assets;
    s.totalSupplyShares -= shares;
    ...
}
```

[M-02] Orderbook management missing during pool pause/unpause

Severity

Impact: Medium

Likelihood: Medium

Description

The pausePool function only changes the pause state without managing the orderbook:

```
function pausePool(bool pause) external {
    if (msg.sender != IOrderbook(PoolStorage._state().orderBook).owner()) revert
PoolErrors.Unauthorized();
    if (pause) {
        _pause();  // ❌ Missing: _cancelOrders()
    } else {
        _unpause();  // ❌ Missing: _updateOrders()
    }
}
```

When pausing, Orders remain active in the `orderbook`, potentially allowing new interactions but will eventually revert, causing DOS and bad user experience.

This is problematic as the `matchMarketBorrowOrder` or `matchLimitBorrowOrder` doesn't check if the `lender pool` has been paused.

```
            // Memory structure for pool aggregation
            PoolData[] memory poolData = new PoolData[](matchedOrderCount);
            uint256 uniquePoolCount;
            uint256 totalCollateral;

            // 1. Aggregate amounts by pool (O(n^2) but acceptable for small n)
            (poolData, uniquePoolCount) = _aggregatePoolData(matchedOrder);

            // 2. Update collateral fields in poolData using the helper function
            totalCollateral = _calculateAndSetPoolCollateral(poolData, uniquePoolCount,
msg.sender, collateralBuffer);
```

Recommendations

1. Add `order operation` during pause state changes.
2. Add `pool state` check when matching orders.

[M-03] Remove liquidity lacks pool liquidity validation

Severity

Impact: Medium

Likelihood: Medium

Description

In the `removeLiquidity` function, the validation only checks share availability:

```
uint256 shares = ERC4626(_queuePriority[i].pool).previewWithdraw(
    _queuePriority[i].totalAmount
);
if (poolShares[_queuePriority[i].pool] < shares) continue;  // ❌ Only checks vault shares
```

However, the pool's available liquidity is not validated just like `maxWithdrawableAssets`:

```
        for (uint256 i = 0; i < len; i++) {
            uint256 availablePoolAssets = poolAssets(pools[i]);
            (PoolGetter.PoolData memory previewPool, , ,) =
IPoolImplementation(pools[i]).getPoolData();
            uint256 poolAvailableLiquidity = previewPool.totalSupplyAssets -
previewPool.totalBorrowAssets;
            if (poolAvailableLiquidity > availablePoolAssets) {
                assets += availablePoolAssets;
            } else {
                assets += poolAvailableLiquidity;
            }
        }
```

The problem is that `previewWithdraw` returns the shares needed for the requested amount. But the pool may not have enough available liquidity to fulfill the withdrawal. This means that: **The vault has shares, but the pool doesn't have assets to redeem them**, thus the withdrawal will fail at the pool level.

Take the following example: - Vault has 1000 shares in Pool A (worth 1000 USDC). - Pool A has 500 USDC available liquidity (500 USDC borrowed). - Manager tries to remove 800 USDC. - previewWithdraw(800) returns 800 shares (vault has enough). - But pool only has 500 USDC available. - Finally, the withdrawal fails at pool level.

Recommendations

Add pool liquidity validation in `removeLiquidity`.

[M-04] Inconsistent seize cap between collateral and share-based liquidation

Severity

Impact: Medium

Likelihood: Medium

Description

The `_liquidate` function handles two different input methods:

1. When specifying collateral amount (`seizedAssets` > 0):

```
uint256 actualSeized = seizedAssets > seizeCap ? seizeCap : seizedAssets;
uint256 seizedQuoted = actualSeized.mulDiv(
    collateralPrice,
    PoolConstants.ORACLE_PRICE_SCALE,
    Math.Rounding.Ceil
);
repaidShares = seizedQuoted
    .mulDiv(PoolConstants.WAD, bonusFactor, Math.Rounding.Ceil)
    .toSharesUp(totalBorrowAssets, totalBorrowShares);
seizedAssets = actualSeized; // Correctly capped
```

1. When specifying share amount (`repaidShares` > 0):

```
uint256 rawSeize = repaidShares
    .toAssetsDown(totalBorrowAssets, totalBorrowShares)
    .mulDiv(bonusFactor, PoolConstants.WAD)
    .mulDiv(PoolConstants.ORACLE_PRICE_SCALE, collateralPrice);
seizedAssets = rawSeize > seizeCap ? seizeCap : rawSeize; // Caps seized but doesn't adjust
repaidShares
```

When using share-based liquidation, if `rawSeize` exceeds `seizeCap`, the function caps `seizedAssets` but does not recalculate `repaidShares` based on the actual seized amount. This means the liquidator repays more debt than they should for the collateral they actually receive.

Example: - Liquidator wants to repay 100 shares. - This would normally require 100 ETH collateral (with a bonus). - But the seize cap is 95 ETH based on dynamic. - Function caps seized collateral to 95 ETH but still repays 100 shares. - Liquidator gets 95 ETH but repays debt equivalent to 100 ETH.

This is not related to slippage control, but the lack of update, causing the user to pay more.

Recommendations

When capping the seized amount in share-based liquidation, recalculate the repaid shares based on the actual seized collateral.

[M-05] Interest accrual during pause blocks repayment, causing liquidations

Severity

Impact: Medium

Likelihood: Medium

Description

The `pausePool` function allows the `orderbook` owner to pause and unpause the pool:

```
function pausePool(bool pause) external {
    if (msg.sender != IOrderbook(PoolStorage._state().orderBook).owner()) revert
PoolErrors.Unauthorized();
    if (pause) {
        _pause();
    } else {
        _unpause();
    }
}
```

When paused, all user functions (including `repay`) are blocked by the `whenNotPaused` modifier:

```
function repay(
    uint256 assets,
    uint256 shares,
    address onBehalf
) external whenNotPaused returns (uint256, uint256) { // <-- Blocked when paused
```

```
    PoolStorage.PoolState storage s = PoolStorage._state();
    // ...
}
```

The Problem: - Interest Continues to Accrue: Every time `accrueInterest()` is called (which happens on any state change), interest is added to `totalBorrowAssets` and `totalSupplyAssets` based on how much time has passed. - No Repayment Possible: Users cannot call `repay()` during the pause to reduce their debt. - Unfair Liquidations: When the pool is unpaused, borrowers may find their positions immediately liquidatable due to accumulated interest, even if they had sufficient collateral before the pause.

Recommendations

Allow the user to join the `repay` queue during the pause, and add the `repay queue` amount after the pause is resumed.

[M-06] Incorrect fee share calculation in preview function uses stale supply assets

Severity

Impact: Low

Likelihood: High

Description

The `_previewAccrueInterest` function contains an inconsistency in fee share calculations. While the function correctly updates `previewPool.totalSupplyAssets` with accrued interest, it incorrectly uses the stale `s.totalSupplyAssets` value when calculating manager and protocol fee shares.

```
    function _previewAccrueInterest(PoolStorage.PoolState storage s) internal view returns
(PoolData memory previewPool, uint256 previewApy) {
        ...
        previewPool.totalBorrowAssets += accruedInterest;
        previewPool.totalSupplyAssets += accruedInterest;  // Updated value

        ...

        if (s.managerFee != 0) {
            managerFeeAmount = accruedInterest.mulDiv(s.managerFee, PoolConstants.WAD);
            uint256 managerFeeShares = managerFeeAmount.mulDiv(s.totalSupplyShares,
s.totalSupplyAssets - managerFeeAmount); // Uses stale s.totalSupplyAssets instead of updated
previewPool.totalSupplyAssets
            previewPool.totalSupplyShares += managerFeeShares;
        }

        protocolFeeAmount =
accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(), PoolConstants.WAD);
```

```
        uint256 protocolFeeShares = protocolFeeAmount.mulDiv(s.totalSupplyShares,
s.totalSupplyAssets - protocolFeeAmount);
        previewPool.totalSupplyShares += protocolFeeShares;
        ...
    }
```

It leads to incorrect fee calculation and `totalSupplyShares`. Vaults relying on `poolAssets` (which uses `previewRedeem`) will have incorrect share valuations, potentially affecting deposit/withdrawal/asset allocation calculations.

Recommendations

It's recommended to use the updated `previewPool.totalSupplyAssets` value instead of the stale `s.totalSupplyAssets`.

[M-07] Heap violation in partial fills breaks liquidity matching priority

Severity

Impact: High

Likelihood: Low

Description

The vulnerability occurs when the largest order (at the root of the heap, `entries[0]`) is only partially filled. The code correctly reduces its liquidity via `entry.amount -= fill;`. However, after this modification, the order's `amount` has decreased, and it is very likely no longer the largest in the heap. The function fails to call a "heapify-down" operation to restore the max-heap property.

As a result, the heap is left in an inconsistent state for the next transaction. A subsequent taker will match against the element at `entries[0]`, which is no longer guaranteed to be the order with the most liquidity.

```
        entry.amount = entry.amount - fill; // liquidity decreased

        if (entry.amount == 0) {
            tree.removeEntry(compositeKey, i); // removeEntry correctly re-heapifies
            entriesCount--;

            if (entriesCount == 0) {
                nodeRemoved = true;
                break;
            }
        } else {
            i++; // no heapify.
        }
```

It's important to note that this bug is masked when the lender is a standard AvonPool. After the borrow function is called on the pool, it automatically triggers `_updateOrders`, which cancels and requotes all its orders on the book. This `_cancelPoolOrders` call effectively removes the partially-filled order, causing the heap to be correctly re-organized.

The protocol's design, however, is open. Any contract that implements the `IPoolImplementation` interface can act as a lender. If these lenders are not required to implement the same auto-requote logic, it will leave the heap in a corrupted state.

Recommendations

The OrderbookLib should be responsible for maintaining its own state integrity, regardless of the behavior of its callers. Calling `_heapifyDown` when an entry's amount is reduced.

[M-08] Heap invariant violation in `removeEntry` causes order mismatch

Severity

Impact: Medium

Likelihood: Medium

Description

The `RedBlackTreeLib.removeEntry` function contains a flaw that breaks the max-heap invariant. This can lead to the order matching engine failing to select the best available offer for a borrower, violating the core liquidity-time priority principle of the order book.

The function attempts to remove an element from the heap. This involves replacing the element to be deleted with the last element in the heap array and then shrinking the array. The current implementation only ever calls `_heapifyDown`. It completely omits the necessary check and corresponding call to `_heapifyUp` when the replacement element is larger than its new parent.

It happens when `cancelBorrowOrder` is called.

```
    function removeEntry(Tree storage tree, uint256 compositeKey, uint256 index) internal {
        ...
        if (index != lastIndex) {
            tree.entriesMap[compositeKey][index] = tree.entriesMap[compositeKey][lastIndex];
            delete tree.entriesMap[compositeKey][lastIndex];
            tree.entryCount[compositeKey] = lastIndex;

            // Maintain heap property if needed
            if (lastIndex > 0) {
>>>                 _heapifyDown(tree, compositeKey, index);
            }
        } else {
```

```
            delete tree.entriesMap[compositeKey][lastIndex];
            tree.entryCount[compositeKey] = lastIndex;
        }

        // If no more entries for this key, remove the node
        if (tree.entryCount[compositeKey] == 0) {
            tryRemove(tree, compositeKey);
        }
    }
```

Consider a valid max-heap of lender orders and the removal of a non-root element.

- Initial Valid Heap: Let's assume the heap array is `[100, 80, 90, 70, 60, 50, 85]`. This is a valid max-heap.
- Operation: An order at `index = 4` (amount `60`) is canceled, and `removeEntry` is called.
- Swap and Pop: The last element at `index = 6` (amount `85`) is swapped into the position at `index = 4`. The heap array becomes `[100, 80, 90, 70, 85, 50]`.
- Invariant Violation: The new element `85` at `index = 4` is now checked against its parent at `index = (4-1)/2 = 1` (amount `80`). Since `85 > 80`, the max-heap property is violated upwards. The element `85` should be sifted up.
- Failure: The flawed code proceeds to call `_heapifyDown` on `index = 4`. As this node has no children, the function does nothing. The heap remains corrupted.

As a result, the element at the root of the heap (index 0) may no longer be the actual highest-priority order, causing the matching engine to provide suboptimal execution for borrowers.

Recommendations

The `removeEntry` function must be modified to correctly restore the heap property by choosing whether to sift the replacement element up or down. After swapping the last element into the deleted position, a comparison with the element's new parent is required.

[M-09] `_repayWithExactShares` does not update state, miscalculating collateral

Severity

Impact: Medium

Likelihood: Medium

Description

The _repayWithExactShares function fails to update the borrower's position state after repayment, unlike the standard `_repay` function. Specifically, it doesn't update:

- s.positions[onBehalf].poolBorrowAssets.
- s.positions[onBehalf].poolBorrowShares.
- s.positions[onBehalf].updatedAt.

```
    function _repayWithExactShares(
        PoolStorage.PoolState storage s,
        uint256 shares,
        address onBehalf
    ) internal returns (uint256, uint256) {
        ...
        s.positions[onBehalf].borrowShares -= shares; // @audit Missing position state updates
        s.totalBorrowShares -= shares;
        s.totalBorrowAssets = s.totalBorrowAssets > assets ?
            s.totalBorrowAssets - assets :
            0;

        ...
    }
```

This causes `previewBorrow` to use stale pool totals when calculating collateral requirements for subsequent borrows, potentially leading to incorrect collateral amounts in orderbook matching.

Recommendations

Update the position state variables consistently with the _repay function:

```
function _repayWithExactShares(
    PoolStorage.PoolState storage s,
    uint256 shares,
    address onBehalf
) internal returns (uint256, uint256) {
    // ... existing logic ...

    s.positions[onBehalf].borrowShares -= shares;
    s.totalBorrowShares -= shares;
    s.totalBorrowAssets = s.totalBorrowAssets > assets ?
        s.totalBorrowAssets - assets :
        0;

+   s.positions[onBehalf].poolBorrowAssets = s.totalBorrowAssets;
+   s.positions[onBehalf].poolBorrowShares = s.totalBorrowShares;
+   s.positions[onBehalf].updatedAt = s.lastUpdate;

    // ... rest of function ...
}
```

[M-10] Missing `accrueInterest()` call before updating manager and protocol fees

Severity

Impact: Medium

Likelihood: Medium

Description

`_executeUpdateManagerFee()` and `_executeUpdateProtocolFee()` are both called by the timelock contract to apply fee updates. For simplicity, we'll focus on `_executeUpdateManagerFee()`, but the same logic applies to `_executeUpdateProtocolFee()`.

```
function _executeUpdateManagerFee(
        uint64 newFee
    ) external {
        if (msg.sender != address(this)) revert PoolErrors.Unauthorized();

        PoolStorage.PoolState storage s = PoolStorage._state();
        uint64 oldFee = s.managerFee;
        s.managerFee = newFee;

        emit PoolEvents.ManagerFeeUpdated(address(this), oldFee, newFee);
    }
```

As you can see, `accrueInterest()` is not called before updating the fees. As a result, the next `accrueInterest()` operation incorrectly applies the new pool fee to the entire period since the last update. This causes the interest accrued during that period to be miscalculated.

Recommendations

To resolve the issue, call `accrueInterest()` before applying the new fees.

```
function _executeUpdateManagerFee(
        uint64 newFee
    ) external {
        if (msg.sender != address(this)) revert PoolErrors.Unauthorized();

+       accrueInterest();

        PoolStorage.PoolState storage s = PoolStorage._state();
        uint64 oldFee = s.managerFee;
        s.managerFee = newFee;

        emit PoolEvents.ManagerFeeUpdated(address(this), oldFee, newFee);
    }
```

```
function _executeUpdateProtocolFee(
        uint64 newFee
    ) external {
        if (msg.sender != address(this)) revert PoolErrors.Unauthorized();

+       accrueInterest();

        PoolStorage.PoolState storage s = PoolStorage._state();
        uint64 oldFee = s.protocolFee;
        s.protocolFee = newFee;

        emit PoolEvents.ProtocolFeeUpdated(address(this), oldFee, newFee);
    }
```

Note: To implement this solution, the visibility of the `accrueInterest()` function must be changed from `external` to `public`.

[M-11] `managerFee` and `protocolFeeAmount` bypassed with frequent `accrueInterest()`

Severity

Impact: Medium

Likelihood: Medium

Description

`accrueInterest()` can be called by anyone to account for the interest accrued. This function internally calls `_accrueInterest()` to perform the calculations.

```
function _accrueInterest(PoolStorage.PoolState storage s) internal returns (uint256
managerFeeShares, uint256 protocolFeeShares) {
        uint256 currentTime = block.timestamp;
@>      uint256 elapsed = currentTime - s.lastUpdate;
        if (elapsed == 0) return (0, 0);

///code...

        // Calculate manager fee shares if applicable
        if (managerFee != 0 && accruedInterest > 0) {
@>          managerFeeAmount = accruedInterest.mulDiv(managerFee, PoolConstants.WAD);
            managerFeeShares = managerFeeAmount.mulDiv(totalSupplyShares, totalSupplyAssets -
managerFeeAmount);
            totalNewShares += managerFeeShares;
        }

        // Calculate protocol fee shares if there's interest
        if (accruedInterest > 0) {
@>          protocolFeeAmount =
accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(), PoolConstants.WAD);
            protocolFeeShares = protocolFeeAmount.mulDiv(totalSupplyShares, totalSupplyAssets -
```

```
protocolFeeAmount);
            totalNewShares += protocolFeeShares;
        }

        // Only update if there are new shares
        if (totalNewShares > 0) {
            totalSupplyShares += totalNewShares;
        }

///code...
    }
```

The fees are calculated based on the time `elapsed` and `totalBorrowAssets`. If a malicious user calls `accrueInterest()` after a very short time, the `managerFeeAmount` and `protocolFeeAmount` may round down to zero when calculated, meaning that even though some interest has accrued, these fees are not properly accounted for. This allows the fees to be effectively bypassed.

This is problematic because an attacker can repeatedly call this function without actually interacting with the protocol (e.g., depositing, withdrawing, repaying), and the attack is particularly effective on MegaETH due to its low gas costs. This enables the attacker to call the function at very short intervals, preventing the fees from ever being collected.

To better understand the issue, copy the following POC into `BorrowRepayTest.t.sol`. To view the logs, it is necessary to add logging inside `_accrueInterest()`.

```
function testAvoid_managerFeeAmountprotocolFeeAmount() public {
        uint256 borrowAmount = 500e6;
        uint256 collateralAmount = pool.previewBorrow(borrower, borrowAmount,
DEFAULT_COLLATERAL_BUFFER);

        vm.startPrank(borrower);

        // Deposit collateral
        collateralToken.approve(address(pool), collateralAmount);
        pool.depositCollateral(collateralAmount, borrower);

        // Borrow
        uint256 balanceBefore = loanToken.balanceOf(borrower);
        (uint256 assets, uint256 shares) = pool.borrow(
            borrowAmount,
            0,
            borrower,
            borrower,
            borrowAmount
        );

        // Verify results
        assertEq(assets, borrowAmount, "Should receive requested borrow amount");
        assertGt(shares, 0, "Should receive borrow shares");
        assertEq(loanToken.balanceOf(borrower), balanceBefore + borrowAmount, "Loan token
balance should increase");

        // Verify position
        _verifyPosition(borrower, borrowAmount, collateralAmount);
```

```
        vm.stopPrank();

        //attacker makes calls in a short period of time
        vm.warp(block.timestamp + 30);
        pool.accrueInterest();
    }
```

```
----------LOGS----------
accruedInterest: 16
managerFeeAmount: 0
protocolFeeAmount: 0
```

Recommendations

To solve the problem, only allow calling `accrueInterest()` after a minimum time has elapsed.

[M-12] `removeQueueEntry` may incorrectly reset `depositHead`

Severity

Impact: Medium

Likelihood: Medium

Description

When `removeQueueEntry` is called, it shifts the queues from the specified index and removes the last entry in the `queue`.

```
    function removeQueueEntry(uint256 index, QueueType _queueType) external onlyVaultManager {
        PriorityEntry[] storage queue = _queueType == QueueType.Deposit ? _queue.depositQueue :
_queue.withdrawQueue;
        uint256 lastIndex = queue.length - 1;
        if (queue.length == 0 || index >= queue.length) revert IncorrectInput();
        for (uint256 i = index; i < lastIndex; i++) {
            queue[i] = queue[i + 1];
        }
        queue.pop();
        if (_queueType == QueueType.Deposit) {
>>>         if (_queue.depositHead >= queue.length) {
                _queue.depositHead = 0;
            }
        } else {
>>>         if (_queue.withdrawHead >= queue.length) {
                _queue.withdrawHead = 0;
            }
        }
        emit QueueEntryRemoved(index, _queueType);
    }
```

However, it incorrectly resets `depositHead` / `withdrawHead` when it reaches `queue.length`. Consider a scenario where there are three queue entries (0, 1, 2), and `depositHead` is currently pointing to index 2. When `removeQueueEntry` is called to remove entry 1, the queue becomes (0, 2). Since `depositHead` is now greater than or equal to `queue.length`, it incorrectly resets to 0.

Recommendations

Before checking `depositHead` and `withdrawHead`, if either is greater than index, decrease it by 1.

```
    function removeQueueEntry(uint256 index, QueueType _queueType) external onlyVaultManager {
        PriorityEntry[] storage queue = _queueType == QueueType.Deposit ? _queue.depositQueue :
_queue.withdrawQueue;
        uint256 lastIndex = queue.length - 1;
        if (queue.length == 0 || index >= queue.length) revert IncorrectInput();
        for (uint256 i = index; i < lastIndex; i++) {
            queue[i] = queue[i + 1];
        }
        queue.pop();

+       if (_queue.depositHead > index) {
+           _queue.depositHead--;
+       }

        if (_queueType == QueueType.Deposit) {
            if (_queue.depositHead >= queue.length) {
                _queue.depositHead = 0;
            }
        } else {
            if (_queue.withdrawHead >= queue.length) {
                _queue.withdrawHead = 0;
            }
        }
        emit QueueEntryRemoved(index, _queueType);
    }
```

[M-13] `MAX_LIMIT_ORDERS` check in `insertLimitBorrowOrder` may block valid updates

Severity

Impact: Medium

Likelihood: Medium

Description

`insertLimitBorrowOrder` checks if the caller's `orders.length` is equal to or greater than `MAX_LIMIT_ORDERS`. if so, the operation reverts.

```
    function insertLimitBorrowOrder(
        uint64 rate,
        uint64 ltv,
        uint256 amount,
        uint256 minAmountExpected,
        uint256 collateralBuffer,
        uint256 collateralAmount
    ) external nonReentrant whenNotPaused {
        if (amount == 0) revert ErrorsLib.ZeroAssets();
>>>     if (borrowersOrders[msg.sender].length >= MAX_LIMIT_ORDERS) revert
ErrorsLib.MaxOrdersLimit();
        if (collateralBuffer < 0.01e18) revert ErrorsLib.InvalidInput();

        BorrowerLimitOrder[] storage orders = borrowersOrders[msg.sender];
>>>     if (orders.length >= MAX_LIMIT_ORDERS) revert ErrorsLib.MaxOrdersLimit();

        IERC20(orderbookConfig.collateral_token).safeTransferFrom(msg.sender, address(this),
collateralAmount);

        uint256 refundAmount;
        for (uint256 i; i < orders.length; ++i) {
            if (orders[i].rate == rate && orders[i].ltv == ltv) {
                refundAmount = orders[i].collateralAmount;
                _cancelBorrowerOrder(msg.sender, rate, ltv, orders[i].amount, i);
                break;
            }
        }

        // Insert new order
        borrowerTree._insertOrder(false, rate, ltv, amount);
        orders.push(
            BorrowerLimitOrder(rate, ltv, amount, minAmountExpected, collateralBuffer,
collateralAmount)
        );

        // Refund collateral from canceled order if any
        if (refundAmount > 0) {
            IERC20(orderbookConfig.collateral_token).safeTransfer(msg.sender, refundAmount);
        }

        emit EventsLib.BorrowOrderPlaced(msg.sender, rate, ltv, amount, minAmountExpected);
    }
```

However, this could incorrectly cause a revert if the user provides an existing `rate` and `ltv` pair, as it removes the previous order and creates a new one, effectively updating the existing order.

Thus, reverting here prevents users from updating their orders.

Recommendations

Consider moving the check to the end of the operation.

```
    function insertLimitBorrowOrder(
        uint64 rate,
        uint64 ltv,
        uint256 amount,
        uint256 minAmountExpected,
        uint256 collateralBuffer,
        uint256 collateralAmount
    ) external nonReentrant whenNotPaused {
        if (amount == 0) revert ErrorsLib.ZeroAssets();
-        if (borrowersOrders[msg.sender].length >= MAX_LIMIT_ORDERS) revert
ErrorsLib.MaxOrdersLimit();
        if (collateralBuffer < 0.01e18) revert ErrorsLib.InvalidInput();

        BorrowerLimitOrder[] storage orders = borrowersOrders[msg.sender];
-        if (orders.length >= MAX_LIMIT_ORDERS) revert ErrorsLib.MaxOrdersLimit();

       // ....

        // Refund collateral from canceled order if any
        if (refundAmount > 0) {
            IERC20(orderbookConfig.collateral_token).safeTransfer(msg.sender, refundAmount);
        }

+        if (orders.length > MAX_LIMIT_ORDERS) revert ErrorsLib.MaxOrdersLimit();

        emit EventsLib.BorrowOrderPlaced(msg.sender, rate, ltv, amount, minAmountExpected);
    }
```

[M-14] Large pools fail to quote up to 99% of their available liquidity

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol's core value proposition relies on pools quoting their available liquidity on the central orderbook to create deep, competitive lending markets. However, an error in `LiquidityAllocator.getQuoteSuggestions` causes large pools to quote only 1% of their available liquidity, leaving 99% of capital idle.

In `getQuoteSuggestions`, the size of each order (`liquidityPerTick`) is calculated by dividing the `availableLiquidity` by `tickCount`. However, the function then proceeds to create only `quoteSuggestions` (max 10) orders. Therefore:

Total liquidity quoted = `quoteSuggestions × (availableLiquidity / tickCount)`

The issue stems from a logical mismatch between two variables: - `tickCount`: Divides the pool's liquidity range into discrete steps (up to 1,000 for pools >$100M). - `quoteSuggestions`: The actual number of orders created (hard-capped at 10).

```
    function getQuoteSuggestions(
        PoolStorage.PoolState storage s,
        uint16 tickCount,
        uint256 availableLiquidity
    ) internal view returns (uint64[] memory rates, uint256[] memory liquidity) {
        uint16 quoteSuggestions = tickCount > 10 ? PoolConstants.QUOTE_SUGGESTIONS :
tickCount; // max 10
        rates = new uint64[](quoteSuggestions);
        liquidity = new uint256[](quoteSuggestions);
        uint256 currentUtilization = availableLiquidity == 0 ? PoolConstants.MAX_UTILIZATION :
(s.totalBorrowAssets * PoolConstants.MAX_UTILIZATION) / s.totalSupplyAssets;

        uint256 utilizationPerTick = (PoolConstants.MAX_UTILIZATION - currentUtilization) /
tickCount;
        uint256 liquidityPerTick = availableLiquidity / tickCount; // Divides by up to 1,000
        ...
    }
```

For a large pool:

- tickCount = 1,000.
- quoteSuggestions = 10.
- Total quoted = 10 × (liquidity / 1,000) = 1% of available liquidity.

The interest rate for each band is also wrongly calculated: it's not mapped to the actual change in utilization that borrowing that band would cause. It's mapped to a hypothetical, much larger change. This makes the quoted curve shape unrepresentative of the pool's actual IRM for the liquidity being offered.

It leads to 90-99% of the capital in large pools sitting idle, earning no interest and providing no value to the ecosystem, directly harming the returns for the pool's LPs.

Recommendations

To use `quoteSuggestions` as the divisor when calculating the size of each liquidity band, instead of `tickCount`:

```
function getQuoteSuggestions(
    PoolStorage.PoolState storage s,
    uint16 tickCount,
    uint256 availableLiquidity
) internal view returns (uint64[] memory rates, uint256[] memory liquidity) {
    uint16 quoteSuggestions = tickCount > 10 ? PoolConstants.QUOTE_SUGGESTIONS : tickCount;
    rates = new uint64[](quoteSuggestions);
    liquidity = new uint256[](quoteSuggestions);
    uint256 currentUtilization = availableLiquidity == 0 ? PoolConstants.MAX_UTILIZATION :
(s.totalBorrowAssets * PoolConstants.MAX_UTILIZATION) / s.totalSupplyAssets;

-   uint256 utilizationPerTick = (PoolConstants.MAX_UTILIZATION - currentUtilization) /
tickCount;
-   uint256 liquidityPerTick = availableLiquidity / tickCount;
+   uint256 utilizationPerTick = (PoolConstants.MAX_UTILIZATION - currentUtilization) /
quoteSuggestions;
```

```
+   uint256 liquidityPerTick = availableLiquidity / quoteSuggestions;

    for (uint256 i; i < quoteSuggestions; ++i) {
        // ...
    }
}
```

[M-15] Lack of safety buffer between borrow and liquidation thresholds

Severity

Impact: Medium

Likelihood: Medium

Description

In both the `borrow` and `liquidation` logic, the protocol uses `_isPositionSafe` as the only condition:

File: src/pool/extensions/BorrowRepay.sol

```
if (!s._isPositionSafe(onBehalf)) revert PoolErrors.InsufficientCollateral();
```

```
if (s._isPositionSafe(borrower)) revert PoolErrors.HealthyPosition();
```

The problem is that when a user borrows, the protocol allows them to borrow up to the exact limit where `_isPositionSafe` returns true. Any small change (e.g., interest accrual, oracle price update, or even a rounding error) can immediately flip their position to `unsafe`, making it eligible for liquidation.

This creates a poor user experience and increases the risk of "griefing" or accidental liquidations, as users have no buffer zone to react to market changes.

Recommendations

Introduce a safety buffer (e.g., a small percentage below the liquidation threshold) when checking if a user can borrow.

Low findings

[L-01] `previewBorrow` missing available liquidity validation

The `previewBorrow` function calculates the collateral required for a borrow without verifying if the pool has sufficient available liquidity.

```
function previewBorrow(
    PoolStorage.PoolState storage s,
    address borrower,
    uint256 assets,  // Requested borrow amount
    uint256 collateralBuffer
) internal view returns (uint256 collateralAmount) {
    // ... collateral price and interest accrual ...

    uint256 collateralRequired = assets.mulDiv(PoolConstants.ORACLE_PRICE_SCALE,
collateralPrice, Math.Rounding.Ceil)
        .mulDiv((PoolConstants.WAD + collateralBuffer), lltv, Math.Rounding.Ceil);

    // ❌ Missing: Check if pool has enough available liquidity
    // uint256 availableLiquidity = previewPool.totalSupplyAssets -
previewPool.totalBorrowAssets;
    // if (assets > availableLiquidity) return 0; // or revert
}
```

This can lead to situations where users are told they can borrow an amount that the pool cannot actually provide, resulting in failed borrow transactions.

[L-02] Existing whitelisted pools lack `IRM` validation after `IRM` disabling

The `OrderbookFactory.setIrm()` function allows the owner to enable/disable Interest Rate Models (IRMs):

```
function setIrm(address irm, bool status) external onlyOwner {
    OrderbookFactoryStorage.FactoryState storage s = OrderbookFactoryStorage._state();
    if (irm == address(0)) revert ErrorsLib.InvalidInput();
    if (s.isIRMEnabled[irm] == status) revert ErrorsLib.AlreadySet();
    s.isIRMEnabled[irm] = status;
    emit EventsLib.SetIrm(irm, status);
}
```

When whitelisting a pool, the system validates that the pool's IRM is enabled:

```
if (!IOrderbookFactory(ORDERBOOK_FACTORY).isIRMEnabled(IPoolImplementation(pool).getIRM())) {
    revert ErrorsLib.IRMNotEnabled();
}
```

However, no validation occurs for existing whitelisted pools when their IRM is disabled. This means: - A pool can be whitelisted when its `IRM` is enabled. - The IRM can later be disabled via `setIrm(irm, false)`. - The pool remains whitelisted and can continue operating with a disabled IRM. - No automatic removal or suspension mechanism exists.

The only way to remove a pool is through manual intervention via `removePool()` or `forceRemovePool()` functions, which require owner action and don't automatically trigger when IRMs are disabled.

[L-03] APY calculation silently fails when result is zero

The APY calculation silently fails when the computed APY equals zero, without any explicit handling or logging.

```
uint256 newApy = ABDKMath64x64.mulu(apyFixed, 1e18);
if (newApy > 0) {
    s.poolApy = newApy;  // ❌ Only updates when APY > 0
}
```

`0 APY` could indicate calculation errors, edge cases, or indicate that the APY calculation is not working as expected. Current silent failure could potentially lead to stale or incorrect APY values being displayed to users.

For example, the previous APY is 5.2%. New calculation results in 0 due to precision loss or something. But APY remains to be 5.2% which could trick users.

[L-04] `RC4626` max functions ignore vault withdrawal constraints

The Vault contract has a sophisticated `maxWithdrawableAssets()` function:

```
function maxWithdrawableAssets() external view returns (uint256) {
    address[] memory pools = totalPools();
    uint256 assets = availableLiquidity();
    uint256 len = pools.length;
    for (uint256 i = 0; i < len; i++) {
        uint256 availablePoolAssets = poolAssets(pools[i]);
        (PoolGetter.PoolData memory previewPool, , ,) =
IPoolImplementation(pools[i]).getPoolData();
        uint256 poolAvailableLiquidity = previewPool.totalSupplyAssets -
previewPool.totalBorrowAssets;
        if (poolAvailableLiquidity > availablePoolAssets) {
            assets += availablePoolAssets;
        } else {
            assets += poolAvailableLiquidity;
        }
    }
    return assets;
}
```

However, the `ERC4626` standard `maxWithdraw` and `maxRedeem` functions are not overridden. This means users calling the standard ERC4626 functions may receive incorrect maximum withdrawal amounts that don't account for the vault's actual liquidity constraints.

[L-05] Withdrawal queue remains uncleared when pool shares are depleted

In the pool management logic:

```
if (poolShares[_queuePriority[i].pool] == 0) {
    _pools.remove(_queuePriority[i].pool);  // Remove from pools set
}
// Missing: Clean up withdrawQueue entries for this pool
```

When a pool's shares are reduced to zero, the pool is removed from the `_pools` set, but the corresponding withdrawal entries in the `withdrawQueue` are not cleaned up. This leaves withdrawal requests that can never be fulfilled, causing unnecessary processing attempts and potential gas waste.

[L-06] Liquidation bonus and seize cap have inconsistent scaling

The liquidation system uses a smooth quadratic scaling for the bonus factor (3% to 12%) but implements a sudden jump in the seize cap (25% to 100%).

```
        if (diff <= PoolConstants.SOFT_RANGE) {
            // Soft band: flat 3% bonus, cap seize to 25% of collateral
            bonusFactor = PoolConstants.BASE_LIQ_BONUS; // 1.03e18
            seizeCap = position.collateral.mulDiv(
                PoolConstants.SOFT_SEIZE_CAP,
                PoolConstants.WAD
            );
        } else {
            // Hard band: quadratic ramp up to 12%
            uint256 quad = diff.mulDiv(diff, PoolConstants.WAD); // (diff)^2/WAD < WAD
            bonusFactor = PoolConstants.BASE_LIQ_BONUS +
                          (PoolConstants.MAX_LIQ_BONUS - PoolConstants.BASE_LIQ_BONUS)
                            .mulDiv(quad, PoolConstants.WAD);
            seizeCap = position.collateral; // full seize allowed
        }
```

This inconsistency creates a discontinuity where liquidators may be incentivized to wait for positions to deteriorate further before liquidating, as the reward increases smoothly, but the ability to seize collateral jumps dramatically.

[L-07] Lack of validation in `borrow()` permits ambiguous parameters

The `AvonPool::borrow` function accepts both `assets` and `shares` parameters but lacks validation to ensure that exactly one of them is greater than zero. This can lead to ambiguous `borrow` requests where both parameters are zero or both are non-zero, potentially causing unexpected behavior or failed transactions.

```
function borrow(
    uint256 assets,
    uint256 shares,
    address onBehalf,
    address receiver,
    uint256 minAmountExpected
) external whenNotPaused returns (uint256, uint256) {
```

Here the branch is based on whether `assets > 0`:

```
(assets, shares) = assets > 0 ? s._borrow(assets, onBehalf, receiver, minAmountExpected):
    s._borrowWithExactShares(shares, onBehalf, receiver, minAmountExpected);
```

- Both Parameters Zero: If both `assets` and `shares` are zero, the function will call `s._borrowWithExactShares(0, ...)`, which likely results in a failed transaction or no borrow occurring.
- Both Parameters Non-Zero: If both `assets` and `shares` are greater than zero, the function will only use the `assets` parameter and ignore the `shares` parameter, which may not be the intended behavior and could confuse users.

It is recommended to add validation at the beginning of the function to ensure exactly one parameter is greater than zero.

[L-08] Redundant logic in liquidity allocation from impossible conditions

The LiquidityAllocator library contains redundant code logic:

```
uint256 startUtil = i == 0 ? currentUtilization : currentUtilization + uint256(i *
utilizationPerTick);
if (startUtil > PoolConstants.MAX_UTILIZATION) {
    liquidity[i] = 0;
    rates[i] = 0;
} else {
    rates[i] = IIrm(s.config.irm).computeBorrowRate(PoolConstants.MAX_UTILIZATION, startUtil);
    liquidity[i] = liquidityPerTick;
}
```

There are 2 issues:

1. Impossible Condition: `startUtil` can never exceed `PoolConstants.MAX_UTILIZATION` because: `currentUtilization` is already bounded by `MAX_UTILIZATION` and `utilizationPerTick` is calculated as `(MAX_UTILIZATION - currentUtilization) / tickCount`. Therefore, `currentUtilization + (i * utilizationPerTick)` will always be `≤ MAX_UTILIZATION`.
2. Redundant Ternary Operator: The expression `i == 0 ? currentUtilization : currentUtilization + uint256(i * utilizationPerTick)` is unnecessarily complex. When `i = 0`, the second part evaluates to `currentUtilization + 0`, which equals `currentUtilization`.

It is recommended to simplify the code by removing the redundant check and ternary operator.

[L-09] Lack of `share == 0` check in `_deposit` function

When a user deposits assets into `AvonPool`, the internal `_deposit` function is called. This function correctly checks that the assets to be deposited are not zero.

```
function _deposit(
    PoolStorage.PoolState storage s,
    uint256 assets,
    address receiver
) internal returns (uint256 shares) {
    if (assets == 0) revert PoolErrors.ZeroAssets(); // Checks that input assets are not zero

    // ... conversion from assets to shares happens here
    shares = assets.toSharesUp(s.totalSupplyAssets, s.totalSupplyShares);

    // Missing check: there is no validation that `shares` is greater than zero

    s.totalSupplyAssets += assets;
    s.totalSupplyShares += shares;
    // ...
}
```

It is strongly recommended to add a check in `_deposit` to ensure that a non-zero amount of shares is minted. Likewise, a check should be added to `_mint` to ensure it corresponds to a non-zero amount of assets. This prevents "donation" attacks and protects users from losing their funds due to precision issues.

[L-10] No slippage control in deposit/withdraw leads to bad exchange rates

The `AvonPool` contract implements the `EIP-4626` standard for its core functions like `deposit`, `mint`, `withdraw`, and `redeem`. The exchange rate between the underlying asset and the pool share is dynamic: **it changes each time interest accrues when other users borrow from the pool.**

```
    function deposit(
        uint256 assets,
        address receiver
    ) public override whenNotPaused returns (uint256 shares) {
        PoolStorage.PoolState storage s = PoolStorage._state();
        accrueInterest(s); // <= HERE
        shares = super.deposit(assets, receiver);
        s._deposit(assets, shares, receiver);
        s._updateOrders();
    }
```

```
// Update pool totals
totalBorrowAssets += accruedInterest;
totalSupplyAssets += accruedInterest;
```

However, the standard `EIP-4626` functions implemented in `AvonPool` do not include a slippage control parameter. This exposes users to price risk: the final amount of shares they receive (or assets they redeem) can be worse than what they anticipated at the time of transaction submission, especially in a volatile or congested network environment.

A user might calculate an expected number of shares based on the state they see, but by the time their transaction is included in a block, preceding transactions (from other users) could have already triggered accrueInterest one or more times. This changes the exchange rate.

To mitigate this risk while maintaining compatibility with the EIP-4626 standard, it is recommended to introduce new, non-standard functions that include a slippage control parameter.

[L-11] EIP-4626: max-functions overlook paused state

The `AvonPool` contract, which implements the EIP-4626 standard, can be paused by an administrator. When paused, all functions that execute deposits or withdrawals (e.g., deposit, mint, withdraw, redeem) correctly revert.

```
    function mint(
        uint256 shares,
        address receiver
    ) public override whenNotPaused returns (uint256 assets) {
        PoolStorage.PoolState storage s = PoolStorage._state();
        accrueInterest(s);
        assets = super.mint(shares, receiver);
```

```
        s._mint(assets, shares, receiver);
        s._updateOrders();
    }
```

However, the corresponding view functions meant to signal the maximum allowable amounts (`maxDeposit` , `maxMint` , etc.) do not reflect this paused state. They continue to return large values instead of zero. This breaks a core requirement of the EIP-4626 standard,

> MUST return the maximum amount of assets `deposit` that would allow to be deposited for `receiver` and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

This can cause integrating protocols to fail and users to waste gas on transactions that are guaranteed to revert.

It is recommended to override the `EIP-4626` `max-functions` in `AvonPool.sol` to account for the contract's paused state. When the pool is paused, these functions should return `0` .

[L-12] Pool tokens with varied fees share same name and symbol

The `AvonPool` contract constructor generates a name and symbol for its pool token by combining the names and symbols of the underlying `loanToken` and `collateralToken` .

```
        ERC20(
          string(abi.encodePacked("Avon ", ERC20(_cfg.loanToken).name(), "/",
ERC20(_cfg.collateralToken).name())),
          string(abi.encodePacked("a",  ERC20(_cfg.loanToken).symbol(),"/",
ERC20(_cfg.collateralToken).symbol()))
        )
```

However, this generation logic omits the pool's `fee` parameter. As a result, **multiple pools managing the same token pair but configured with different fees will produce pool tokens that are indistinguishable by their name and symbol**.

This ambiguity can confuse users and dApp frontends, potentially leading to users interacting with a pool that does not match their fee expectations. While this does not lead to a direct loss of funds from the protocol's perspective, it creates an user experience issue and information gap.

It is recommended to incorporate the pool's fee into the generated name and symbol for the pool token. This will ensure that each pool token is uniquely identifiable and accurately reflects its core characteristics.

```
    ERC20(
      // Append fee to the name, e.g., "Avon USDC/WETH 5" for 0.05%
      string(abi.encodePacked(
          "Avon ",
          ERC20(_cfg.loanToken).name(),
          "/",
```

```
            ERC20(_cfg.collateralToken).name(),
            " ",
            Strings.toString(_fee) // Assuming fee is in bps
        )),
        // Append fee to the symbol, e.g., "aUSDC/WETH-5"
        string(abi.encodePacked(
            "a",
            ERC20(_cfg.loanToken).symbol(),
            "/",
            ERC20(_cfg.collateralToken).symbol(),
            "-",
            Strings.toString(_fee)
        ))
    )
```

[L-13] Inconsistent share to asset conversion in position safety check

The `_isPositionSafe` function calculates borrowed assets using a manual formula that doesn't account for virtual shares, while the rest of the protocol consistently uses the `SharesLib` functions that include virtual shares and assets for precision and manipulation resistance.

```
    function _isPositionSafe(
        PoolStorage.PoolState storage s,
        address borrower
    ) internal view returns (bool) {
        ...
        uint256 assetRatio = s.totalBorrowAssets.mulDiv(PoolConstants.WAD, s.totalBorrowShares,
Math.Rounding.Ceil);
        uint256 borrowedAssets = position.borrowShares * assetRatio;

        ...
    }
```

It's recommended to use `SharesLib.toAssetsUp` for consistency with the rest of the protocol.

[L-14] Interest overestimated in `_previewAccrueInterest()` due to quote buffer

In `PoolGetter._previewAccrueInterest`, the elapsed time calculation includes a buffer. Its inclusion in general the calculations creates interest inaccuracies.

```
    function _previewAccrueInterest(PoolStorage.PoolState storage s) internal view returns
(PoolData memory previewPool, uint256 previewApy) {
        uint256 elapsed = block.timestamp + PoolConstants.QUOTE_VALID_PERIOD - s.lastUpdate; //
@audit +10 seconds buffer
        ...
    }
```

It leads to: - Vaults calculating total assets via `poolAssets` receive inflated values, leading to the wrong share price. - Vault fee accrual is based on an inflated gain calculation.

It's recommended to calculate the accurate interest using `elapsed = block.timestamp - s.lastUpdate`.

[L-15] Incorrect APY calculation from updated total supply assets

The APY calculation in both `AccrueInterest._accrueInterest` and `PoolGetter._previewAccrueInterest` functions incorrectly uses the updated `totalSupplyAssets` (which includes accrued interest) in the denominator when calculating the `lenderRate`. This leads to an understated APY calculation.

In the interest accrual process, the totalSupplyAssets is updated to include the newly accrued interest:

```
    function _accrueInterest(PoolStorage.PoolState storage s) internal returns (uint256
managerFeeShares, uint256 protocolFeeShares) {
        ...
        totalSupplyAssets += accruedInterest; // Update pool totals - this modifies
totalSupplyAssets
        ...
        if (totalSupplyAssets > 0 && elapsed > 0 && accruedInterest > 0) {
            uint256 accruedInterestWithFees = accruedInterest;
            if (managerFeeAmount > 0 || protocolFeeAmount > 0) {
                accruedInterestWithFees -= (managerFeeAmount + protocolFeeAmount);
            }
            int128 lenderRate = ABDKMath64x64.div(
                ABDKMath64x64.fromUInt(accruedInterestWithFees),
>>>             ABDKMath64x64.fromUInt(totalSupplyAssets * elapsed) // @audit Uses updated
totalSupplyAssets
            );
            int128 lnValue = ABDKMath64x64.ln(ABDKMath64x64.add(lenderRate,
ABDKMath64x64.fromUInt(1)));
            int128 exponent = ABDKMath64x64.mul(lnValue, ABDKMath64x64.fromUInt(31536000));
            int128 apyFixed = ABDKMath64x64.sub(ABDKMath64x64.exp(exponent),
ABDKMath64x64.fromUInt(1));
            uint256 newApy = ABDKMath64x64.mulu(apyFixed, 1e18);
            if (newApy > 0) {
                s.poolApy = newApy;
            }
        }
        ...
    }
```

It's recommended to use `totalSupplyAssets` before interest to calculate `lenderRate`.

[L-16] Advertised borrow rate can be misleadingly low

The `_updateOrders` function quotes liquidity bands on the `Orderbook` using the interest rate calculated at the *start* of that band's utilization. This can create a significant disconnect between the advertised rate and the final executed rate if a single borrow transaction pushes the pool's utilization past a critical threshold, such as the "kink" in the Interest Rate Model (IRM).

```
    function getQuoteSuggestions(
        PoolStorage.PoolState storage s,
        uint16 tickCount,
        uint256 availableLiquidity
    ) internal view returns (uint64[] memory rates, uint256[] memory liquidity) {
        ...
        for (uint256 i; i < quoteSuggestions; ++i) {
            uint256 startUtil = i == 0 ? currentUtilization : currentUtilization + uint256(i *
utilizationPerTick);
            if (startUtil > PoolConstants.MAX_UTILIZATION) {
                liquidity[i] = 0;
                rates[i] = 0;
            } else {
                rates[i] = IIrm(s.config.irm).computeBorrowRate(PoolConstants.MAX_UTILIZATION,
startUtil); // @audit Rate calculated at band start
                liquidity[i] = liquidityPerTick;
            }
        }
    }
```

A borrower may be attracted by a low advertised rate, only to find that their own transaction causes the pool's effective rate to jump dramatically. While the protocol is functioning correctly to protect itself from high utilization, this can lead to a poor and surprising user experience due to severe, unexpected interest rate slippage.

```
    function insertLimitBorrowOrder(
        uint64 rate, // @audit actual rate can be significantly different
        uint64 ltv,
        uint256 amount,
        uint256 minAmountExpected,
        uint256 collateralBuffer,
        uint256 collateralAmount
    ) external nonReentrant whenNotPaused {
```

Consider a scenario:

- Pool has $100 total supply, $0 borrowed (0% utilization), 1 tick configured (all liquidity in single band).
- The orderbook shows an attractive ~2% APR (typical rate at 0% utilization), this pool appears first in the orderbook due to the low advertised rate.
- User attempts to borrow $100 through a limit order at 2% rate, which represents 100% of available liquidity. Transaction pushes utilization from 0% to 100% instantly, the borrower pays the high rate despite seeing 2% advertised.

It is recommended that users have rate slippages when creating a limit order.

[L-17] Lack of timelock in `increaseLLTV()` allows immediate risk changes

The `increaseLLTV` function executes immediately upon call:

```
    function increaseLLTV(
        uint64 newLTV
    ) external onlyRole(PROPOSER_ROLE) {
        if (newLTV < PoolConstants.MIN_LTV || newLTV > PoolConstants.MAX_LTV) revert
PoolErrors.InvalidInput();
        if (newLTV <= PoolStorage._state().config.lltv) revert PoolErrors.InvalidInput();
        PoolStorage.PoolState storage s = PoolStorage._state();
        uint64 oldLTV = s.config.lltv;
        s._cancelOrders();
        s.config.lltv = newLTV;
        s._updateOrders();

        emit PoolEvents.LLTVIncreased(address(this), oldLTV, newLTV);
    }
```

LLTV is a critical risk parameter that determines when borrower positions become liquidatable. Increasing LLTV allows borrowers to take on higher leverage against their collateral, which directly increases the risk profile of the lending pool. It doesn't allow time for lenders to withdraw funds if they disagree with the new risk level.

Lenders face an increased risk of losses without adequate notice to adjust their positions.

It's recommended to implement timelock governance for `increaseLLTV` consistent with other parameter changes.

[L-18] Inefficient order matching can lead to revert

The `_matchOrder` function in OrderbookLib continues to iterate through the entire order book even when it's mathematically impossible to find more matches, leading to unnecessary gas consumption and potential reverts due to gas limits.

When matching borrower orders against the lender tree, the function walks through orders from lowest to highest interest rates (since lender orders are sorted by ascending rate via `tree.first`). However, once a lender's rate exceeds the borrower's maximum acceptable rate, no subsequent lenders can possibly match since they will all have even higher rates.

```
    function _matchOrder(RedBlackTreeLib.Tree storage tree, bool isLender, uint64 rate, uint64
ltv, uint256 amount)
        internal
        returns (MatchedOrder memory matchedOrders)
    {
        ...

        bytes32 currentPtr = tree.first();
```

```
        while (remaining > 0 && currentPtr != bytes32(0)) {
            if (gasleft() < MIN_REMAINING_GAS) break;

            bool recordDetails = matchedCount < MAX_MATCH_DETAILS;
            bytes32 nextPtr = RedBlackTreeLib.next(currentPtr);

            uint256 originalKey = RedBlackTreeLib.value(currentPtr);
            bool nodeRemoved = false;

            if (_isMatchingOrder(currentPtr, rate, ltv, isLender)) {
               ...
            }

            ...
            currentPtr = nextPtr; // Continues even when no more matches are possible
    }
```

For borrower matching against lender tree:

- `nodeRate <= searchRate` : Lender's rate must be ≤ borrower's maximum rate.
- Since lenders are sorted by ascending rate, once `nodeRate > searchRate`, all subsequent lenders will also have `nodeRate > searchRate`.

```
    function _isMatchingOrder(bytes32 ptr, uint256 searchRate, uint256 searchLTV, bool isLender)
        private
        view
        returns (bool)
    {
        (uint256 nodeRate, uint256 nodeLTV) =
RedBlackTreeLib._unpackCompositeKey(RedBlackTreeLib.value(ptr));

        return isLender
            ? type(uint64).max - uint64(nodeRate) >= searchRate && uint64(nodeLTV) <= searchLTV
            : uint64(nodeRate) <= searchRate && type(uint64).max - uint64(nodeLTV) >= searchLTV;
    }
```

It's recommended to add early termination logic when rate-based matching becomes impossible.

[L-19] Silent failure in order cancellation when there's no match

The protocol maintains a borrower's limit order in two places: the `borrowersOrders` array and the `borrowerTree` for matching. When `_cancelBorrowerOrder` is called, it iterates through the tree to find and remove the order. If the for loop completes without finding a matching entry, the function exits silently instead of reverting.

If, for any reason, an attacker can trigger a state where their order is in the `borrowersOrders` array but not the tree, then call `cancelBorrowOrder`. The function will succeed, refunding their collateral without updating any state. It can lead to a drain of all collateral in the orderbook.

```
    function _cancelBorrowerOrder(address borrower, uint256 rate, uint256 ltv, uint256 amount,
uint256 index)
        internal
    {
        ...

        for (uint256 i; i < entryCount; ++i) {
            RedBlackTreeLib.Entry storage entry = RedBlackTreeLib.getEntryAt(borrowerTree,
compositeKey, i);
            if (entry.account == borrower) {  // @audit doesn't revert if no match
                borrowerTree._cancelOrder(false, compositeKey, i);
                if (orderAmount - amount != 0) {
                    borrowerTree._insertOrder(false, uint64(rate), uint64(ltv), orderAmount -
amount);
                    order.amount -= amount;
                    order.collateralAmount -= collateralAmount;
                    order.minAmountExpected -= minExpected;
                } else {
                    if (index != ordersLength - 1) {
                        orders[index] = orders[ordersLength - 1];
                    }
                    orders.pop();
                }
                break;
            }
        }
    }
```

It's recommended to defensively revert if there's no match in `_cancelBorrowerOrder`.

[L-20] Missing `nonReentrant` modifier in `cancelBorrowOrder` function

The `cancelBorrowOrder` function lacks the `nonReentrant` modifier, creating an inconsistency with other similar functions in the contract and potentially exposing the system to reentrancy risks.

```
    function cancelBorrowOrder(uint256 rate, uint256 ltv, uint256 amount, uint256 index)
external {
        if (amount == 0) revert ErrorsLib.ZeroAssets();
        BorrowerLimitOrder memory order = borrowersOrders[msg.sender][index];
        uint256 collateralAmount = order.collateralAmount.mulDivDown(amount, order.amount);

        _cancelBorrowerOrder(msg.sender, rate, ltv, amount, index);

        IERC20(orderbookConfig.collateral_token).safeTransfer(msg.sender, collateralAmount);
        emit EventsLib.BorrowOrderCanceled(msg.sender, rate, ltv, amount);
    }
```

[L-21] Users cannot control collateral requirements in market borrow orders

The `matchMarketBorrowOrder` function does not provide users with sufficient control over their collateral requirements, potentially forcing them to accept unfavorable LTV ratios that require excessive collateral.

In the current implementation of matchMarketBorrowOrder, users can only specify the desired borrow amount and the minimum amount they expect to receive. However, users cannot control the maximum collateral they're willing to provide or the maximum LTV ratio they're willing to accept. The function automatically matches against any available orders regardless of their LTV requirements using the best available lender rate.

```
    function matchMarketBorrowOrder(uint256 amount, uint256 minAmountExpected, uint256
collateralBuffer)
        external
        nonReentrant
        whenNotPaused
    {
        if (amount == 0) revert ErrorsLib.ZeroAssets();
        if (collateralBuffer < 0.01e18) revert ErrorsLib.InvalidInput();

        (uint64 rate,) = lenderTree._getBestLenderRate();
        uint64 ltv = OrderbookLib.MIN_LTV; // @audit set to minimum LTV

        MatchedOrder memory matchedOrder = lenderTree._matchOrder(false, rate, ltv, amount);

        ...

        // 3. Single collateral transfer
        IERC20 collateralToken = IERC20(orderbookConfig.collateral_token);
        collateralToken.safeTransferFrom(msg.sender, address(this), totalCollateral); // @audit
cannot control the totalCollateral

        ...
    }
```

It may lead to users being forced to provide significantly more collateral than anticipated.

It's recommended to allow users to set a maximum collateral amount or a maximum LTV requirement.

[L-22] `totalAssets` function can revert due to gas limit exceeded when managing many pools

The `totalAssets` function in the Vault contract iterates through all pools and performs complex calculations for each. It calls `ERC4626(poolAddress).previewRedeem`, which in turn executes `_previewAccrueInterest`, containing expensive operations. If the vault manages a large number of pools, the function can potentially revert due to out-of-gas errors. It can lead to no user being able to deposit, mint, withdraw, or redeem funds.

```
function totalAssets() public view override returns (uint256) {
    address[] memory pools = totalPools();
    uint256 assets = ERC20(asset()).balanceOf(address(this));
    uint256 len = pools.length;
    for(uint256 i = 0; i < len; i++) { // loop through each pool and call `previewRedeem`
        uint256 shares = poolShares[pools[i]];
        if (shares > 0) {
            assets += poolAssets(pools[i]);
        }
    }
    return assets;
}
```

It's recommended to add a maximum number of pools constraint.

[L-23] Access bypass via `TimelockController` direct interaction

Critical admin functions such as `updateProtocolFee` and `updateTimeLockDuration` in AvonPool that are restricted to `DEFAULT_ADMIN_ROLE` can be bypassed by accounts with `PROPOSER_ROLE` through direct calling `TimelockController.schedule`.

```
function updateProtocolFee(
    uint64 newFee
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    ...
}

function updateTimeLockDuration(
    uint64 newDuration
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    ...
}
```

It is because `TimelockController.schedule` only requires `PROPOSER_ROLE`, not `DEFAULT_ADMIN_ROLE`.

```
function schedule(
    address target,
    uint256 value,
    bytes calldata data,
    bytes32 predecessor,
    bytes32 salt,
    uint256 delay
) public virtual onlyRole(PROPOSER_ROLE) {
    bytes32 id = hashOperation(target, value, data, predecessor, salt);
    _schedule(id, delay);
    emit CallScheduled(id, 0, target, value, data, predecessor, delay);
    if (salt != bytes32(0)) {
        emit CallSalt(id, salt);
    }
}
```

This vulnerability allows a less privileged role (Pool Manager) to unilaterally change a critical financial parameter (protocol fee) after a time delay, bypassing the intended control of the main protocol admin.

It's recommended to override `TimelockController.schedule` to extract function selector from calldata to restrict admin-only functions.

[L-24] `AvonPool`'s withdrawal and `borrow` operations are vulnerable to griefing

In the current `AvonPool` implementation, operations within the same block do not accrue interest, and both `borrow` and `repay` operations are allowed. This opens an attack vector: an attacker with sufficient collateral could borrow all remaining liquidity, enabling them to grief legitimate withdrawal operations and `borrow` requests.

Consider preventing same-block `borrow` and `repay` operations.

[L-25] Missing maximum amount check borrowing with shares

When `borrow` is called, it allows users to provide `assets` / `shares` and `minAmountExpected`. However, when `_borrowWithExactShares` is used, it's also possible that the provided shares result in a borrowed asset amount greater than expected.

```
    function _borrowWithExactShares(
        PoolStorage.PoolState storage s,
        uint256 shares,
        address onBehalf,
        address receiver,
        uint256 minAmountExpected
    ) internal returns (uint256, uint256) {
        if (shares == 0) revert PoolErrors.InvalidInput();
        if (receiver == address(0)) revert PoolErrors.ZeroAddress();
        if (!s._isSenderPermitted(onBehalf)) revert PoolErrors.Unauthorized();

>>>     uint256 assets = shares.toAssetsDown(s.totalBorrowAssets, s.totalBorrowShares);
        s.positions[onBehalf].borrowShares += shares;
        s.totalBorrowShares += shares;
        s.totalBorrowAssets += assets;
        s.positions[onBehalf].poolBorrowAssets = s.totalBorrowAssets;
        s.positions[onBehalf].poolBorrowShares = s.totalBorrowShares;
        s.positions[onBehalf].updatedAt = s.lastUpdate;

        if (!s._isPositionSafe(onBehalf)) revert PoolErrors.InsufficientCollateral();
        if (s.totalBorrowAssets > s.totalSupplyAssets) revert
PoolErrors.InsufficientLiquidity();
>>>     if (assets < minAmountExpected) revert PoolErrors.InsufficientAmountReceived();

        SafeERC20.safeTransfer(ERC20(s.config.loanToken), receiver, assets);

        emit PoolEvents.Borrow(
            address(this),
```

```
            msg.sender,
            onBehalf,
            receiver,
            assets,
            shares
        );

        return (assets, shares);
    }
```

Consider allowing users to check the maximum amount expected.

[L-26] Pausing repay may cause liquidation risk when pool is unpaused

`AvonPool` can be paused, affecting all operations, including `repay`. During the paused state, interest continues to accrue over time, which may cause users to become liquidatable once the pool is unpaused.

Consider triggering accrue interest when the pool is paused, and updating `s.lastUpdate` to the latest timestamp without triggering accrue interest when unpause is called. This would effectively pause interest accrual while `AvonPool` is paused.

[L-27] `totalCount` may exceed counterParty length causing revert

When `_matchOrder` is called, either from `matchMarketBorrowOrder` or `matchLimitBorrowOrder`, it iterates through `lenderTree` orders and matches them with the borrower's request.

```
    function _matchOrder(RedBlackTreeLib.Tree storage tree, bool isLender, uint64 rate, uint64
ltv, uint256 amount)
        internal
        returns (MatchedOrder memory matchedOrders)
    {
        if (amount == 0) revert ErrorsLib.InvalidInput();

        matchedOrders.counterParty = new address[](MAX_MATCH_DETAILS);
        matchedOrders.amounts = new uint256[](MAX_MATCH_DETAILS);
        uint256 matchedCount = 0;
        uint256 totalMatchCount = 0;
        uint256 remaining = amount;

        bytes32 currentPtr = tree.first();
        while (remaining > 0 && currentPtr != bytes32(0)) {
           // ...

            if (_isMatchingOrder(currentPtr, rate, ltv, isLender)) {
                uint256 compositeKey = RedBlackTreeLib.value(currentPtr);
                uint256 entriesCount = tree.getEntryCount(compositeKey);

                uint256 i = 0;
                while (i < entriesCount && remaining > 0) {
```

```
                    RedBlackTreeLib.Entry storage entry = tree.getEntryAt(compositeKey, i);
                    uint256 fill = entry.amount > remaining ? remaining : entry.amount;

>>>                 if (recordDetails) {
                        matchedOrders.counterParty[matchedCount] = entry.account;
                        matchedOrders.amounts[matchedCount] = fill;
                        matchedCount++;
                        recordDetails = matchedCount < MAX_MATCH_DETAILS;
                    }

                    matchedOrders.totalMatched += fill;
                    remaining -= fill;
>>>                 totalMatchCount++;

                    emit EventsLib.OrderMatched(
                        isLender ? msg.sender : entry.account, isLender ? entry.account :
msg.sender, rate, ltv, fill
                    );

                    // ...

>>>     matchedOrders.totalCount = totalMatchCount;
    }
```

The issue is that while the length of `counterParty` is restricted by `MAX_MATCH_DETAILS`, `matchedOrders.totalCount` is not, it continues to grow until all `remaining` is filled or no more `currentPtr` entries are available.

This could cause a revert, when `_aggregatePoolData` is called, it iterates through `matchedOrder_.totalCount` and accesses `counterParty` in each iteration. If `matchedOrder_.totalCount` exceeds the length of `counterParty`, the operation will revert.

Consider also checking `recordDetails` inside the while loop, if it's false, exit the loop.

[L-28] Inconsistent interest calculations lead to inflated APY

The interest accrual uses continuous compounding:

```
int128 expResult = ABDKMath64x64.exp(expInput); // e^(r*t)
uint256 accruedInterest = totalBorrowAssets.mulDiv(expFactor - PoolConstants.WAD,
PoolConstants.WAD);
```

But the APY calculation uses simple interest to derive the rate:

```
int128 lenderRate = ABDKMath64x64.div(
    ABDKMath64x64.fromUInt(accruedInterestWithFees),
    ABDKMath64x64.fromUInt(totalSupplyAssets * elapsed)  // ❌ Simple interest formula
);
```

Problem: - Interest accrual: Interest = Principal * (e^(r*t) - 1) (continuous compounding). - APY rate derivation: r = Interest / (Principal * time) (simple interest).

For the same interest amount, **the simple interest rate is higher than the continuous compounding rate, leading to inflated APY calculations**.

Take the following example: - Principal: 1000 USDC. - Interest: 2 USDC. - Time: 1 day (86400 seconds).

According to the Simple interest rate(current calculation), the rate is:

```
r = 2 / (1000 * 86400) ≈ 2.3148148148148148e-08
math.exp(r*31536000) = 2.0750806076741224
```

So APY is 107.5%.

But for the correct one:

```
r = ln(1 + 2/1000) / 86400 ≈ 2.3125030817975207e-08
math.exp(r*31536000) = 2.0735683668508913
```

So APY is 107.3%.

Recommendations

Use continuous compounding to derive the rate for APY calculation.

[L-29] Unbounded loop in `_cancelPoolOrders()` may cause out-of-gas

In `Orderbook.sol`, the `_cancelPoolOrders` function contains unbounded nested loops that can consume excessive gas:

```
function _cancelPoolOrders(address pool, uint256 ltv) internal {
    uint256[] memory orders = poolOrders[pool];
    uint256 ordersLength = orders.length;

    if (ordersLength > 0) {
        for (uint256 i; i < ordersLength; ++i) {  // Outer loop - unbounded
            uint256 currentIR = orders[i];
            uint256 currentLTV = type(uint64).max - ltv;

            uint256 compositeKey = RedBlackTreeLib._packCompositeKey(currentIR, currentLTV);
            uint256 entryCount = RedBlackTreeLib.getEntryCount(lenderTree, compositeKey);

            for (uint256 j; j < entryCount; ++j) {  // Inner loop - unbounded
                RedBlackTreeLib.Entry storage entry = RedBlackTreeLib.getEntryAt(lenderTree,
compositeKey, j);
                if (entry.account == pool) {
                    lenderTree._cancelOrder(true, compositeKey, j);
                    break;
                }
            }
        }
```

```
        delete poolOrders[pool];
    }
}
```

However, transactions can revert with out-of-gas error when processing pools with many orders. And the function is called during critical operations like `removePool()` and `forceRemovePool()`, but no fallback mechanism exists if the operation fails due to gas limits.

Recommendations

Revise the relevant code.

[L-30] Fee checked only at proposal time lets total fee exceed max limit

The fee update functions validate that the new fee plus the current other fee does not exceed `MAX_TOTAL_FEE` only during the proposal phase, but not during execution.

```
// In setManagerFee proposal
if (newFee + getProtocolFee() > PoolConstants.MAX_TOTAL_FEE) revert PoolErrors.InvalidInput();

// In setProtocolFee proposal
if (newFee + PoolStorage._state().managerFee > PoolConstants.MAX_TOTAL_FEE) revert
PoolErrors.InvalidInput();
```

This allows a scenario where multiple fee changes can be proposed and executed in sequence, resulting in a total fee that exceeds the intended maximum limit of 25%.

1. Current fees: managerFee = 10%, protocolFee = 10% (total: 20%).
2. Propose managerFee = 5% (5% + 10% = 15% ✓).
3. Propose managerFee = 14% (14% + 10% = 24% ✓).
4. Execute managerFee = 5% (now managerFee = 5%, protocolFee = 10%).
5. Propose protocolFee = 19% (5% + 19% = 24% < 25% ✓).
6. Execute managerFee = 14%.
7. Execute managerFee = 19% (now it's 14% + 19% > 25%).

As a result, total fees can exceed the intended 25% maximum.

Recommendations

Add validation during execution to ensure the final state respects the maximum fee limit.

[L-31] Insufficient incentives for small liquidations risk bad debt

The liquidation bonus calculation in `Liquidation.sol` uses a percentage-based system:

```
if (diff <= PoolConstants.SOFT_RANGE) {
    // Soft band: flat 3% bonus, cap seize to 25% of collateral
    bonusFactor = PoolConstants.BASE_LIQ_BONUS; // 1.03e18 (3%)
    seizeCap = position.collateral.mulDiv(
        PoolConstants.SOFT_SEIZE_CAP,
        PoolConstants.WAD
    );
} else {
    // Hard band: quadratic ramp up to 12%
    uint256 quad = diff.mulDiv(diff, PoolConstants.WAD); // (diff)^2/WAD
    bonusFactor = PoolConstants.BASE_LIQ_BONUS +
                  (PoolConstants.MAX_LIQ_BONUS - PoolConstants.BASE_LIQ_BONUS)
                    .mulDiv(quad, PoolConstants.WAD);
    seizeCap = position.collateral; // full seize allowed
}
```

The Problem: 1. Gas Cost vs. Reward: For small liquidation amounts, a 3-12% bonus may not cover the gas costs of the liquidation transaction, especially during high gas periods. 2. Example Scenario: - Small position: 100 USDC debt, 150 USDC collateral. - 3% bonus = 3 USDC. - Gas cost: 50-100 USDC (during congestion). - Net loss for liquidator: 47-97 USDC.

1. Bad Debt Accumulation: If liquidators are not incentivized to liquidate small positions, these positions may remain unliquidated even when they become unsafe, leading to bad debt that affects the entire protocol.

Recommendation

Implement a hybrid incentive system that combines a fixed minimum bonus with a percentage-based reward. Or don't allow small positions.

[L-32] Incorrect slippage control in `borrow` function

In the `borrow` function, the slippage control is implemented as:

```
function borrow(
    uint256 assets,
    uint256 shares,
    address onBehalf,
    address receiver,
    uint256 minAmountExpected  // User-provided
) external whenNotPaused returns (uint256, uint256) {
    // ...
    (assets, shares) = assets > 0 ? s._borrow(assets, onBehalf, receiver, minAmountExpected):
        s._borrowWithExactShares(shares, onBehalf, receiver, minAmountExpected);
}
```

And in the internal `_borrow` function:

```
if (assets < minAmountExpected) revert PoolErrors.InsufficientAmountReceived();
```

Both `assets` and `minAmountExpected` are **user-provided parameters**. A user can set `minAmountExpected` to any value they want, making this check meaningless.

In fact, the `_borrow` function should be checking if the actual execution result (the `shares` received) is within acceptable bounds, not comparing user inputs.

Recommendations

Implement proper slippage control.

[L-33] Stale roles persist for former owners due to direct role assignment

When creating new `AvonPool` and `Vault` instances, the system assigns administrative roles to the current `owner` of the `orderbook` or `orderbookFactory`.

For example, in `AvonPoolFactory.sol`, the `orderbook owner` is made a proposer and executor, granting it significant control.

```
// ...
proposers[0] = msg.sender; // Pool manager
proposers[1] = Ownable(orderbook).owner(); // Orderbook owner
executors[0] = msg.sender; // Pool manager
executors[1] = Ownable(orderbook).owner(); // Orderbook owner
// ...
```

Similarly, roles like `CANCELLER_ROLE` and general `_admin` privileges are granted based on this owner address at the time of creation.

```
File: src/pool/AvonPool.sol
// ...
_grantRole(CANCELLER_ROLE, _admin);
// ...
```

The core issue is that this owner address is stored separately within each created `AvonPool` or `Vault` instance. If the owner of the `orderbookFactory` / `orderbook` is transferred from address `A` to `B`, address `A` will retain its admin-level access on all contracts instances created before the role transfer. Same works for `poolManager` as it's previlege could be revoked.

There could be 2 primary risks:

1. If owner A's private key is ever compromised, the attacker gains administrative control over a potentially large number of previously deployed contracts.
2. It's not easy to revoke `A`'s permissions all at once, and this has to be set for every single contract instance created.

NOTE: The same issue exists for `Orderbook` instance creation.

Recommendations

Introduce a layer of indirection for access control. This ensures that permissions can be managed centrally and updated efficiently.

[L-34] Pool position loss in orderbook due to order reinsertion on liquidity changes

The Avon protocol's orderbook system has a vulnerability where pools lose their queue position when updating orders, allowing malicious actors to manipulate borrowing flow to competing pools.

When any liquidity change occurs in a pool (including borrowing), the protocol:

- Calls `_updateOrders` which cancels all existing orders and reinserts them.
- The reinsertion uses the new, higher interest rate.

```
function batchInsertOrder(uint64[] calldata irs, uint256[] calldata amounts)
    external
    whenNotPaused
{
    ...
    uint256 ltv = IPoolImplementation(msg.sender).getLTV();
    _cancelPoolOrders(msg.sender, ltv);

    for (uint256 i; i < length; ++i) {
        if (amounts[i] == 0) break;
        if (length > i+1 && irs[i+1] <= irs[i]) revert ErrorsLib.OrdersNotOrdered();
        lenderTree._insertOrder(true, irs[i], uint64(ltv), amounts[i]);
        poolOrders[msg.sender].push(irs[i]);
    }
}
```

Since the orderbook's Red-Black tree sorts lender orders by lowest interest rate first, even a minimal borrow (1 wei) forces a pool to a higher rate node, place it behind the other orders.

A malicious actor can:

- Monitor for incoming large borrow transactions.
- Front-run with a minimal borrow (1 wei) from competing pools.
- Force those pools to accrue interest and move to higher rate nodes.
- Redirect the large borrower to their preferred pool at the original, lower rate.

Consider a scenario: A malicious actor is a liquidity provider or beneficiary of pool B. Three pools A, B, C offer the same rate and LTV.

Before manipulation: - Node 0: Rate=317097919, LTV=80%, Pool A: 40M, Pool B: 40M, Pool C: 40M - Pool A places first.

After borrowing 1 wei from Pool A: - Node 0: Rate=317097919, LTV=80%, Pool B: 40M, Pool C: 40M. - Node 1: Rate=317097927, LTV=80%, Pool A: 40M (higher rate due to interest).

Borrowers will now match with Pools B and C first at the lower rate (317097919), while Pool A is pushed to a less competitive position (317097927).

This vulnerability allows an attacker to systematically divert borrowing demand away from targeted pools, directly reducing the yield earned by LPs and the fees collected by the pool manager in those pools.

Recommendations

It's recommended to have the minimum borrowing amount.

[L-35] Out of gas issue in `_performWithdraw` function

The `_performWithdraw` function contains errors that can lead to infinite loops and gas exhaustion, making withdrawals impossible under certain conditions.

- The function uses a while loop with the condition `while (toWithdraw > 0 && len > 0)`, where `len` is a static value set at the beginning. When pools cannot provide sufficient liquidity, the function can enter an infinite loop:
  - User requests withdrawal of X tokens.
  - All pools in the withdrawal queue either have insufficient shares (`poolShares[e.pool] < shares`) or no available liquidity (`poolAvailableLiquidity == 0`).
    - It can happen due to `previewWithdraw` rounds up the shares.
    - utilization rate = 100%.
  - Each pool gets skipped, but `toWithdraw` remains unchanged, the transaction runs until it hits the gas limit and reverts.
- The vault has to loop through multiple rounds through the queue to withdraw all. And each `pool.withdraw` call involves complex operations including interest accrual, position updates, and orderbook interactions, making it gas-expensive. It can lead to the gas needed being more than the block gas limit, and the transaction reverts.

```
    function _performWithdraw(uint256 assets) internal {
        uint256 toWithdraw = assets;
        uint256 len = _queue.withdrawQueue.length;

        while (toWithdraw > 0 && len > 0) { // @audit infinite loop condition
            PriorityEntry storage e = _queue.withdrawQueue[_queue.withdrawHead];
            // Resetting remaining if we wrapped around
            if (e.remaining == 0) {
                e.remaining = e.totalAmount;
            }

            uint256 delta = toWithdraw < e.remaining ? toWithdraw : e.remaining;
            (PoolGetter.PoolData memory previewPool, , ,) =
IPoolImplementation(e.pool).getPoolData();
            // Check if the pool has enough assets to withdraw\
            uint256 poolAvailableLiquidity = previewPool.totalSupplyAssets -
previewPool.totalBorrowAssets;
            delta = delta < poolAvailableLiquidity ? delta : poolAvailableLiquidity;
            // Withdraw the assets from the pool
            uint256 shares = ERC4626(e.pool).previewWithdraw(delta); // @audit round up lead to
```

```
poolShares[e.pool] < shares
            if (poolShares[e.pool] < shares || poolAvailableLiquidity == 0) {
                e.remaining = 0;
                _queue.withdrawHead = uint128((_queue.withdrawHead + 1) %
_queue.withdrawQueue.length);
                continue; // @audit skips pool, but toWithdraw remains unchanged
            }

            // Attempt the withdrawal, move to next pool if it fails
            try ERC4626(e.pool).withdraw(delta, address(this), address(this)) returns (uint256
_shares) {
            ...
        }
    }
```

Recommendations

- Refactor the function so that instead of skipping pools with insufficient shares, withdraw what's available
- Consider to limit the number of managed pools.

[L-36] In `cancelBorrowOrder()`, user is forced to cancel minimum loan amount

`cancelBorrowOrder()` allows users to cancel their outstanding limit borrow orders.

```
function cancelBorrowOrder(uint256 rate, uint256 ltv, uint256 amount, uint256 index) external {
        if (amount == 0) revert ErrorsLib.ZeroAssets();
        BorrowerLimitOrder memory order = borrowersOrders[msg.sender][index];
        uint256 collateralAmount = order.collateralAmount.mulDivDown(amount, order.amount);

        _cancelBorrowerOrder(msg.sender, rate, ltv, amount, index);

        IERC20(orderbookConfig.collateral_token).safeTransfer(msg.sender, collateralAmount);
        emit EventsLib.BorrowOrderCanceled(msg.sender, rate, ltv, amount);
    }
```

When the collateral token has fewer decimals than the loan token, the user is forced to cancel a sufficiently large amount of loan tokens in order to receive any collateral back. This happens because `mulDivDown()` is used to compute the proportional collateral, and if the amount being cancelled is too small, the result rounds down to 0, leaving `collateralAmount = 0`.

For example, assume the collateral token has 6 decimals and the loan token has 18 decimals.

1. Bob creates a limit borrow order using `insertLimitBorrowOrder()` with `amount = 100_000e18` and `collateralAmount = 1000e6`.
2. Bob wants to partially cancel the order and calls `cancelBorrowOrder()` with `amount = 1000e9`.
3. The collateral amount is calculated as: `collateralAmount = 1000e6 * 1000e9 / 100_000e18 = 0`.

4. The loan token amount is reduced by `1000e9`, but Bob receives no collateral back.

In this case, the user must cancel at least `1000e11` loan tokens to receive any collateral.

Recommendations

To address the issue, scale `order.collateralAmount` up to match the decimals of the loan token during the calculation, and then convert the result back to the decimals of the collateral token afterward.

[L-37] Vault's `_accrueInterest` updates `prevTotal` even when at a loss

In the current implementation of `_accrueInterest` inside the vault, `prevTotal` will be updated regardless of whether there is `gain` or not.

```
    function _accrueInterest() internal {
        uint256 currentAssets = totalAssets();

        // Only calculate fees if this isn't the first accrual and there's a gain
        if (prevTotal > 0 && currentAssets > prevTotal) {
            // Calculate the gain (interest earned)
            uint256 gain = currentAssets - prevTotal;

            // Calculate manager's share of the gains
            uint256 managerFeesAmount = gain.mulDiv(managerFees, 1e18);

            if (managerFeesAmount > 0) {
                uint256 shares = convertToShares(managerFeesAmount);
                if (shares > 0) {
                    _mint(feeRecipient, shares);
                    emit ManagerFeesAccrued(managerFeesAmount, shares);
                }
            }
        }
        // Update the previous total assets amount for next accrual
>>>     prevTotal = currentAssets;
        emit TotalAssetsUpdated(currentAssets);
    }
```

This is incorrect and could result in unnecessary fees. For instance, if `totalAssets` is currently at a loss (lower than `prevTotal`), and later becomes profitable, the increase from the previous loss to the current profit will be incorrectly treated as a `gain`, leading to an inaccurate calculation of `managerFeesAmount`.

Recommendations

Update `prevTotal` only when `currentAssets > prevTotal`.

[L-38] Bad borrowers reclaim collateral on insolvency, lenders absorb loss

`liquidate()` calls `_liquidate()` to liquidate under-collateralized (unhealthy) positions.

```
function _liquidate(
        PoolStorage.PoolState storage s,
        address borrower,
        uint256 seizedAssets,
        uint256 repaidShares,
        uint256 minSeizedAmount,
        uint256 maxRepaidAsset
    ) internal returns (uint256, uint256) {

///code...

        if (position.collateral == 0 && position.borrowShares > 0) {
            badDebtShares = position.borrowShares;
            badDebtAssets = Math.min(
                totalBorrowAssets,
                badDebtShares.toAssetsUp(totalBorrowAssets, totalBorrowShares)
            );
            totalBorrowAssets -= badDebtAssets;
            totalBorrowShares -= badDebtShares;
@>          s.totalSupplyAssets -= badDebtAssets;
            position.borrowShares = 0;
        }

        s.totalBorrowAssets = totalBorrowAssets;
        s.totalBorrowShares = totalBorrowShares;
        position.poolBorrowAssets = totalBorrowAssets;
        position.poolBorrowShares = totalBorrowShares;
        position.updatedAt = s.lastUpdate;

///code...
```

If the position is insolvent (it has 0 `collateral` but still owes `borrowShares`), the resulting bad debt is absorbed by the lenders (`s.totalSupplyAssets -= badDebtAssets`). This creates a problem because malicious borrowers can exploit this to wipe out their bad debt while recovering all of their collateral, causing the lenders to bear the losses.

This situation can occur if the collateral price drops sharply. In such a scenario, a borrower can call `liquidate()` on their own position, effectively liquidating themselves, clearing their debt, and reclaiming the remaining collateral.

This is possible because the liquidation process includes a `bonusFactor` to incentivize liquidators by allowing them to repay fewer shares than the full debt. However, this bonus can be abused to repay the minimum possible shares, leaving the bad debt to the lenders.

```
///code...

if (diff <= PoolConstants.SOFT_RANGE) {
            // Soft band: flat 3% bonus, cap seize to 25% of collateral
```

```
            bonusFactor = PoolConstants.BASE_LIQ_BONUS; // 1.03e18
            seizeCap = position.collateral.mulDiv(
                PoolConstants.SOFT_SEIZE_CAP,
                PoolConstants.WAD
            );
        } else {
            // Hard band: quadratic ramp up to 12%
            uint256 quad = diff.mulDiv(diff, PoolConstants.WAD); // (diff)^2/WAD
            bonusFactor = PoolConstants.BASE_LIQ_BONUS +
                          (PoolConstants.MAX_LIQ_BONUS - PoolConstants.BASE_LIQ_BONUS)
                            .mulDiv(quad, PoolConstants.WAD);
            seizeCap = position.collateral; // full seize allowed
        }

///code...
```

This can be done either by malicious borrowers or even by a well-intentioned liquidator.

To better understand the issue, copy the following POC into `LiquidationTest.t.sol`.

```
function test_BorrowerLiquidateItslefFullPositionWithBadDebt_AllTheDebtIsEatenByLenders()
public {

        vm.prank(owner);
        oracle.setPrice((1e6*1e36)/1e18);

        //IntinitialBorrowerBalance = collateralToken (actual balance of DAI of the borrower) +
Actual collateral in the position
        uint256 initialCollateralTokenBalance = collateralToken.balanceOf(borrower) +
pool.getPosition(borrower).collateral;
        uint256 initalBorrowedAmount = pool.getPosition(borrower).borrowAssets;

        console.log("Initial Borrowed Amount:", initalBorrowedAmount);

        //Liquidate
        vm.startPrank(borrower);
        loanToken.approve(address(pool), DEFAULT_BORROW_AMOUNT);

        uint256 borrowShares = pool.getPosition(borrower).borrowShares;
        //Liquidate using the minimum possible shares by exploiting the bonusFactor
        pool.liquidate(borrower, 0, borrowShares / 3530, 0, DEFAULT_BORROW_AMOUNT);

        //Check borrower position is completely liquidated with bad debt handled
        PoolGetter.BorrowPosition memory position = pool.getPosition(borrower);
        assertEq(position.borrowShares, 0, "Borrow shares should be zero after complete
liquidation");
        assertEq(position.collateral, 0, "Collateral should be zero after complete
liquidation");
        vm.stopPrank();

        uint256 finalCollateralTokenBalance = collateralToken.balanceOf(borrower);
        uint256 finalLoanTokenBalance = loanToken.balanceOf(borrower);

        console.log("Initial Borrower Collateral Token Balance:",
initialCollateralTokenBalance);
        console.log("Final Borrower Collateral Token Balance :", finalCollateralTokenBalance);
        console.log("Final Borrower Loan Token Balance :", finalLoanTokenBalance);
```

```
        console.log("Borrower Collateral Loss:", initialCollateralTokenBalance -
finalCollateralTokenBalance);
        console.log("Borrower Loan Tokens Loss:", initalBorrowedAmount - finalLoanTokenBalance);
    }
```

```
------------LOGS------------
Initial Borrowed Amount: 500000002
badDebtAssets Cleaned: 499858356
repaidAssets: 141644
Initial Borrower Collateral Token Balance: 10000000000000000000
Final Borrower Collateral Token Balance : 10000000000000000000
Final Borrower Loan Token Balance : 499858356
Borrower Collateral Loss: 0
Borrower Loan Tokens Loss: 141646
```

> **Note:** To observe the `badDebtAssets` and the `repaidAssets`, you need to add a `console.log()` inside `_liquidate()`.

As shown in the logs, the initial borrowed amount was 500 USDC, and the bad debt absorbed by the lenders is 499 USDC.

Additionally, the `repaidAssets ≈ Borrower Loan Tokens Loss` (less than one dollar), as these are the assets paid to the protocol to recover the collateral.

Finally, the bad borrower clears the bad debt by using the lenders' funds, holds the full borrowed amount, and recovers the entire collateral.

Recommendations

One solution is to prevent borrowers from liquidating their own positions. However, a better approach would be to implement alternative strategies to effectively manage and clean users bad debt.