Avon Protocol Audit Report

AvonPool and Avon Vault contracts

Prepared by: alix40, Independent Security Researcher

Date: 01.07.2025

Commit: 4e099254cd1e813ca1129d85cc9c0dcfaf81024c

About **Avon**

Avon is a decentralized lending protocol that implements an orderbook-based system for matching lenders and borrowers. The core components include OrderbookFactory for creating and managing orderbooks, Orderbook for managing lending/borrowing orders using red-black trees, and Pool contracts implementing ERC-4626 for managing deposits, borrowing, and liquidations.

About **Alix40**

Alix40 is a leading smart contract security researcher with over 200 high/medium severity findings across major DeFi protocols. Currently serving as a Lead Senior Watson at Sherlock, they have consistently ranked in the top 5 positions in 15+ competitive audit contests, specializing in lending protocols and complex financial primitives.

Scope

Files in scope:

- src/pool/AvonPool.sol
- src/pool/PoolStorage.sol
- src/utils/*
- src/extensions/*
- src/factory/*
- src/libraries/*

Summary of Findings

ID	Title	Fixed
[H01]	The `repayWithExactShares()` doesn't round in favor of the protocol	✅
[H02]	Direct Stealing Yield because we don't account for accrued interest	✅
[H03]	Malicious users could mask pools from the orderbook	✅
[M01]	The liquidityUSD can't be correctly calculated for tokens with 18 decimals	✅
[M02]	toSharesUp in repay	✅
[M03]	`AvonPool.totalAssets()` is not implemented correctly	✅
[M04]	maxWithdraw/maxRedeem doesn't check for available liquidity	➖
[M05]	Users Positions could be instantly liquidated	➖
[M06]	_performWithdraw() is not restricted and infinite loops are possible	➖
[M07]	poolManagers could bypass access control on pause() to lock funds out	✅
[M08]	APY is not calculated correctly in the vault and pool	✅
[M09]	deposits in the vault are DoSed if one of the pools in the queue are paused	✅
[M10]	`AvonPool.previewBorrow()` might calculate the required collateral wrongly in some conditions	✅
[M11]	missing slippage protection on liquidate	✅
[L01]	AvonPool doesn't have a function to change poolManager Address	➖
[L02]	Protocol Fees	➖
[L03]	all the contracts are not upgradeable	➖
[L04]	cancelOrders() spams the pool with empty orders	✅
[L05]	PoolManagers are not able to change the managerFee in AvonPool	➖
[L06]	the `Vault` and the `AvonPool` don't implement any caps/limits on deposits/borrows	➖
[L07]	PoolManagers are able to drain the pool	➖
[L08]	missing stateUpdate for position.poolBorrowAssets,position.poolBorrowShares,position.updatedAt	✅
[L09]	currentUtilization is not calculated correctly in `getQuoteSuggestions()`	✅

High Risk Findings

[H-01] The `repayWithExactShares()` doesn't round in favor of the protocol

Impact: HIGH - Theft of funds through debt reduction without payment. The attacker's debt gets socialized to other borrowers, potentially making their positions unhealthy and leading to liquidations. This attack is particularly dangerous on MegaETH due to low gas costs and high block gas limits.

Description:
The bug is in part of `_repay` and full in `_repayWithExactShares()`. When computing the amount users need to repay for their borrow shares, the function should round in favor of the protocol. Currently, it uses `toAssetsDown()` which rounds down, allowing attackers to reduce their debt without paying the full amount.

Proof of Concept:

```
function _repayWithExactShares(
    PoolStorage.PoolState storage s,
    uint256 shares,
    address onBehalf
) internal returns (uint256, uint256) {
    if (onBehalf == address(0)) revert PoolErrors.ZeroAddress();

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

    if(shares > s.positions[onBehalf].borrowShares) {
        shares = s.positions[onBehalf].borrowShares;
        assets = shares.toAssetsDown(s.totalBorrowAssets,
s.totalBorrowShares);
    }

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

Attack scenario in a loop do the following:

1. borrow an amount x from pool
2. call `repay(0,1,msg.sender)` when the amount to be sent is calculated the 1 share would be rounded to 0 of assets because of `toAssetsDown()`. Effectively the user will reduce their debt by 1 wei without paying anything

```
function toAssetsDown(uint256 shares, uint256 totalAssets, uint256
totalShares) internal pure returns (uint256) {
    return shares.mulDiv(totalAssets + VIRTUAL_ASSETS, totalShares +
```

```
VIRTUAL_SHARES);
}
```

=> `1*(totalAssets+1)/(totalShares +1e6) == 0`

By repeating this attack multiple times, the attacker will be able to borrow debt and close their position without paying anything. This is especially dangerous on MegaETH where gas cost is very low and the block gas limit is very high (2 billion gas units vs. the usual 80 million gas units on Ethereum mainnet).

Recommended Mitigation:
Use `toAssetsUp()` instead in `_repayWithExactShares()` and `_repay()`

Review: Fixed in `d4121064b80a2bdbe8db5fd815674140f16e4b5f`

[H-02] Direct Stealing Yield because we don't account for accrued interest

Impact: HIGH - Stealing of yield from the vault contract

Description:
Direct theft of yield from vault because `totalAssets()` doesn't account for unaccrued interest in pools. If a pool registered in the vault has accrued interest that hasn't been accounted for yet (the `pool_accrueInterest` function is called by interacting with the pool), the current implementation doesn't consider any accrued interest in the individual pool since the last interaction with the pool.

Proof of Concept:

```
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++) {
        uint256 shares = poolShares[pools[i]];
        if (shares > 0) {
            assets += poolAssets(pools[i]);
        }
    }
    return assets;
}
```

```
function poolAssets(address poolAddress) public view returns (uint256) {
    return ERC4626(poolAddress).previewRedeem(poolShares[poolAddress]);
}
```

`previewRedeem` relies on the overridden `convertToAssets` in AvonPool.sol:

```
function _convertToAssets(uint256 shares, Math.Rounding rounding) internal
view override returns (uint256) {
    PoolStorage.PoolState storage s = PoolStorage._state();
    return shares.mulDiv(s.totalSupplyAssets + 1, s.totalSupplyShares + 10
** _decimalsOffset(), rounding);
}
```

Attack Scenario: Assume a vault with only 1 pool for simplicity. No one interacted with the pool for 1 hour and 1k USDC has been accumulated in the pool, but this is not reflected in `previewRedeem`.

1. When a user/attacker tries to deposit in the vault, the `totalAssets()` value is used to calculate the amount of shares to provide for the depositor. This value is **undervalued** and doesn't account for the accrued interest in the pool, resulting in the user getting more shares of the vault than they should.
2. For maximum value extraction, an attacker will monitor pools and the vault until enough yield has been accumulated before any state update to the pools reflecting that change. The attacker takes a large amount as a flash loan, deposits it in the pool, and in the same transaction withdraws (because state update in the pool will happen and yield will be accrued). They will be able to deposit 10K USDC and withdraw 10.1K USDC, repay the flash loan, and keep the yield stolen from the vault LP holders.

Recommended Mitigation:
The `previewRedeem()` in the AvonPool should be overridden to reflect any accrued interest in the pool that has not yet been added to the `totalSupplyAssets`

Review: Fixed in `49d38841c4f1b1fa3d406f3b7c72c45c38d17555`

[H-03] Malicious users could mask pools from the orderbook

Impact: HIGH - Attackers are able to manipulate the lending pool to block borrows from the pool

Description:
The whole issue is caused by the fact that when the pool tries to update its corresponding lending orders on the attached orderbook, it computes the available liquidity by directly using `erc20.balanceOf(address(this))` instead of deriving it using `s.totalSupplyAssets - s.totalBorrowAssets`.

Proof of Concept:

```
function _updateOrders(PoolStorage.PoolState storage s) internal {
    uint16 tickCount = _getTicks(s.config.oracle);
    (uint64[] memory rates, uint256[] memory liquidity) =
s.getQuoteSuggestions(
        tickCount,
        IERC4626(address(this)).totalAssets()
    );
```

Attack scenario:

1. Attacker takes a flash loan of all available liquidity from the pool. (Now `erc20.balanceOf(pool) == 0`)
2. Attacker next deposits 1 wei so that the pool will update its lending orders on the orderbook. Because the pool doesn't have any tokens (0 liquidity), the lenderTree will be filled with 0 orders
3. Repay the flash loan

This way when a user tries to borrow using the orderbook, the lenderTree will contain false values and the user will not be matched with this pool.

Recommended Mitigation:
Use `s.totalSupplyAssets - s.totalBorrowAssets` instead of `erc20.balanceOf(address(this))` to compute available liquidity.

Review: Fixed in `49d38841c4f1b1fa3d406f3b7c72c45c38d17555`

Medium Risk Findings

[M-01] The liquidityUSD can't be correctly calculated for tokens with 18 decimals

Impact: MEDIUM - Incorrect liquidityUSD calculation leading to wrong liquidity pricing

Description:
The expected decimal precision of `liquidityUSD` must be in 18 decimals. For a token with 18 decimals, `totalLiquidity` will be in 18 decimals already. If we multiply `totalLiquidity` by the price from the oracle, we are sure to surpass the 18 decimals and thus the `liquidityUSD` precision would be messed up.

Proof of Concept:

```
function _getTicks(address oracle) internal view returns (uint16
ticksCount){
    uint256 totalLiquidity = IERC4626(address(this)).totalAssets();
    uint256 liquidityUSD = (totalLiquidity *
IOracle(oracle).getLoanToUsdPrice());
```

Recommended Mitigation:
Fix the calculation by dividing by the loanAsset decimals and standardize `getLoanToUsdPrice` to return 18 decimals precision.

Review: Fixed in `92da8a6119e4edb6094ffd3feb0f436397941d23`

[M-02] toSharesUp in repay

Impact: MEDIUM - Limited loss of value for protocol and Liquidity Providers

Description:
When converting from assets to shares we should round up to make sure that we are rounding in favor of the protocol. However currently the rounding is done wrong.

Proof of Concept:

```
function _repay(
    PoolStorage.PoolState storage s,
    uint256 assets,
    address onBehalf
) internal returns (uint256, uint256) {
    if (onBehalf == address(0)) revert PoolErrors.ZeroAddress();

    uint256 shares = assets.toSharesUp(s.totalBorrowAssets,
s.totalBorrowShares);
```

Recommended Mitigation:
To mitigate this issue, we recommend using `toSharesDown` instead in the repay function.

```
function _repay(
    PoolStorage.PoolState storage s,
    uint256 assets,
    address onBehalf
) internal returns (uint256, uint256) {
    if (onBehalf == address(0)) revert PoolErrors.ZeroAddress();

    uint256 shares = assets.toSharesDown(s.totalBorrowAssets,
s.totalBorrowShares);
```

Review: Fixed in `d4121064b80a2bdbe8db5fd815674140f16e4b5f`

[M-03] `AvonPool.totalAssets()` is not implemented correctly

Impact: MEDIUM - The `AvonPool` contract is not EIP-4626 compliant

Description:
The AvonPool doesn't override the `totalAssets()` implementation from ERC4626 OpenZeppelin implementation which is:

```
function totalAssets() public view virtual returns (uint256) {
    return IERC20(asset()).balanceOf(address(this));
}
```

According to the EIP-4626 standard, the `totalAssets()` function should reflect the assets managed by the pool. In this case, it should be `state.totalSupplyAssets`.

Recommended Mitigation:
Override the `totalAssets()` function in AvonPool to return `s.totalSupplyAssets` instead of the default ERC4626 implementation.

Review: Fixed in `9fe53637b8ddaff34ac9d534c7645d1dd7b0f207`

[M-04] maxWithdraw/maxRedeem doesn't check for available liquidity

Impact: MEDIUM - The `AvonPool` and `Vault` contracts is not EIP-4626 compliant

Description:
According to the EIP-4626 standard, the pool `maxWithdraw()` and `maxRedeem()` should reflect the real amount that the user is able to withdraw. Currently the AvonPool contract doesn't override the default implementation and doesn't correctly show/reflect the available liquidity for the pool users.

This is the official text from the EIP-4626 standard:

```
### maxRedeem
> Maximum amount of Vault shares that can be redeemed from the owner
balance in the Vault, through a redeem call.
```

> MUST return the maximum amount of shares that could be transferred from owner through redeem and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

Recommended Mitigation:
When computing the `maxWithdraw()` and `maxRedeem()`, the AvonPool should consider the available liquidity (e.g., the pool liquidity could be 0 if all the assets are borrowed).

Review: Acknowledged

[M-05] Users Positions could be instantly liquidated

Impact: MEDIUM - Users could create positions that are instantly liquidatable

Description:
In contrast with other lending protocols, the AvonPool only tracks one type of LTV that is used to check whether a position is healthy or not. The `liquidate()` and `borrow()` functions both check for the same exact LTV by calling the function `_isPositionSafe()` that checks for this LTV value.

Recommended Mitigation:
As it is the case with other lending protocols, we recommend using 2 separate LTVs for checking whether a position is healthy and whether a position is liquidatable (liquidation_ltv and ltv). This way, we make sure that the users can borrow up to a maximum LTV and there is a safety margin between that LTV and the LTV where the position becomes liquidatable.

Review: Acknowledged

[M-06] _performWithdraw() is not restricted and infinite loops are possible

Impact: MEDIUM - If there is not enough liquidity in the internal pools we will continue to loop until all the gas of the transaction is spent.

Description:
The `_performWithdraw()` function doesn't have any iterations limit while looping through the internal vault.

Proof of Concept:

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

    while (toWithdraw > 0 && len > 0) {
        PriorityEntry storage e =
_queue.withdrawQueue[_queue.withdrawHead];
        uint256 shares = ERC4626(e.pool).previewWithdraw(delta);
```

```
        if (poolShares[e.pool] < shares || poolAvailableLiquidity == 0) {
            e.remaining = 0;
            _queue.withdrawHead = uint128((_queue.withdrawHead + 1) %
_queue.withdrawQueue.length);
            continue;
        }

        // Attempt the withdrawal, move to next pool if it fails
        try ERC4626(e.pool).withdraw(delta, address(this), address(this))
returns (uint256 _shares) {
            if (e.remaining == 0) {
                _queue.withdrawHead = uint128((_queue.withdrawHead + 1) %
_queue.withdrawQueue.length);
            } catch {
                e.remaining = 0;
                _queue.withdrawHead = uint128((_queue.withdrawHead + 1) %
_queue.withdrawQueue.length);
            }
        }
    }
}
```

As shown in the function, if the `pool.withdraw()` fails for any reason we will increment the withdrawHead by 1 and continue infinitely the loop. There is however no restriction on the amount of iterations.

Recommended Mitigation:
To mitigate this issue, we simply need to add a maximum number of iterations to check for in the `_performWithdraw()`

Review: Acknowledged

[M-07] poolManagers could bypass access control on pause() to lock funds out

Impact: MEDIUM - PoolManagers are able to execute privilege escalation on the `AvonPool`

Description:
Per the current design only the orderbook owner() address is able to pause an `AvonPool`. The access control is enforced in the `pausePool()` function:

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

This access control could actually be bypassed because the vault allows the pool manager to switch the address of the orderbook through the vault functions.

Recommended Mitigation:
We recommend that the `updateOrderBook()` function also be accessible by the `orderBook.owner()`

Review: Fixed in avon-core `e8168077f78fc7e71cb572da56cd17ba25436db8` and avon-periphery `7ea8640af6a7094d2185571d828190b1618d148b`

[M-08] APY is not calculated correctly in the vault and pool

Impact: MEDIUM - The Pool/Vault shows wrong APY

Description:
The issue when calculating the lenderRate in `_previewAccrueInterest()` to compute APY we use the accruedInterest.

```
if (previewPool.totalSupplyAssets > 0 && elapsed > 0 && accruedInterest >
0) {
    int128 lenderRate = ABDKMath64x64.div(
        ABDKMath64x64.fromUInt(accruedInterest),
        ABDKMath64x64.fromUInt(previewPool.totalSupplyAssets * elapsed)
    );
```

The problem however is that the `accruedInterest` variable includes the protocol and manager fees. Meaning actually the APY is not accurate as it includes the fees.

Recommended Mitigation:
To reflect the correct APY, simply remove the managerFees and protocolFees from the `accruedInterest`

Review: Fixed in `9a9d2c8b3495194571ae0df73fcbef8bdf9974a3`

[M-09] deposits in the vault are DoSed if one of the pools in the queue are paused

Impact: MEDIUM - DoS on deposits in the vault

Description:
Per the current implementation, if any of the `pool.deposit` operations will revert, the whole `vault.deposit()` will revert.

Proof of Concept:

```
function _allocateDeposit(uint256 assets) internal {
    uint256 toAllocate = assets;
    uint256 steps;
    uint256 len = _queue.depositQueue.length;

    while (toAllocate > 0 && len > 0 && steps < MAX_QUEUE_PROCESSING) {
        PriorityEntry storage e = _queue.depositQueue[_queue.depositHead];
        // Resetting remaining if we wrapped around
```

```
        if (e.remaining == 0) {
            e.remaining = e.totalAmount;
        }

        uint256 delta = toAllocate < e.remaining ? toAllocate :
e.remaining;
        // Deposit the assets into the pool
        ERC20(asset()).safeIncreaseAllowance(e.pool, delta);
        uint256 shares = ERC4626(e.pool).deposit(delta, address(this));
```

Recommended Mitigation:
We simply recommend adding a try-catch clause on pool deposit and continuing if the deposit reverts.

Review: Fixed in
`b898a458aae417927494a79efe4e3dff0151bcae`,`91814396f96bcd6dfb020054d2ff894f15f8fb69`

[M-10] `AvonPool.previewBorrow()` might calculate the required collateral wrongly in some conditions

Impact: MEDIUM - Incorrect collateral requirement calculation, when computing required collateral in the orderbook.

Description:
The issue is when calculating the borrowedDiff:

```
if (borrowShares != 0 && updatedAt != block.timestamp) {
    uint256 borrowedBefore =
borrowShares.mulDiv(s.positions[borrower].poolBorrowAssets,
s.positions[borrower].poolBorrowShares, Math.Rounding.Ceil);
    uint256 borrowedAfter =
borrowShares.mulDiv(previewPool.totalBorrowAssets,
previewPool.totalBorrowShares, Math.Rounding.Ceil);

    uint256 borrowedDiff = borrowedAfter > borrowedBefore ? borrowedAfter -
borrowedBefore : borrowedBefore - borrowedAfter;

    uint256 collateralDelta = borrowedDiff
        .mulDiv(PoolConstants.ORACLE_PRICE_SCALE, collateralPrice,
Math.Rounding.Ceil)
        .mulDiv(PoolConstants.WAD, lltv, Math.Rounding.Ceil);

    collateralAmount = collateralDelta + collateralRequired;
} else {
    collateralAmount = collateralRequired;
}
```

If the `borrowedAfter < borrowedBefore` (e.g., in case of bad debt socialization), we will also increase the required collateral amount by the difference, which is incorrect.

Recommended Mitigation:
We could fix it as follows:

```
if (borrowShares != 0 && updatedAt != block.timestamp) {
    uint256 borrowedBefore =
borrowShares.mulDiv(s.positions[borrower].poolBorrowAssets,
s.positions[borrower].poolBorrowShares, Math.Rounding.Ceil);
    uint256 borrowedAfter =
borrowShares.mulDiv(previewPool.totalBorrowAssets,
previewPool.totalBorrowShares, Math.Rounding.Ceil);

-    uint256 borrowedDiff = borrowedAfter > borrowedBefore ? borrowedAfter
- borrowedBefore : borrowedBefore - borrowedAfter;
+    uint256 borrowedDiff = borrowedAfter > borrowedBefore ? borrowedAfter
- borrowedBefore : 0;

    uint256 collateralDelta = borrowedDiff
        .mulDiv(PoolConstants.ORACLE_PRICE_SCALE, collateralPrice,
Math.Rounding.Ceil)
        .mulDiv(PoolConstants.WAD, lltv, Math.Rounding.Ceil);

    collateralAmount = collateralDelta + collateralRequired;
} else {
    collateralAmount = collateralRequired;
}
```

Review: Fixed in 5b4d27436fdca2868075909364fbc0d1a2834937

[M-11] missing slippage protection on liquidate

Impact: MEDIUM - unprofitable liquidation/ possible loss of funds

Description:
The liquidate function doesn't implement any slippage protection or any way to protect liquidation bots from Market conditions/deviations or Race Conditions. This is especially dangerous, because of the following line:

```
uint256 amountToSeize = Math.min(seizedAssets, position.collateral);
```

Recommended Mitigation:
Add slippage parameters to the liquidate function to protect users from unfavorable price movements. (e.g minSeizedAmount, or maxRepaidAsset)

Review: Fixed in 9270c7ed0f12fcaf36b86a165ba537c114974334

Low Risk Findings

[L-01] AvonPool doesn't have a function to change poolManager Address

Impact: LOW - It is not possible to change the address of the poolManager. This could be problematic in the case that the private keys of the poolManager is compromised, there is no way to update the access.

Description:
The AvonPool contract lacks functionality to update the poolManager address, which could be problematic if the poolManager's private keys are compromised.

Recommended Mitigation:
Allow poolManagers to change the address

Review: Acknowledged

[L-02] Protocol Fees

Impact: LOW - The protocol fee in all Avon pools is hardcoded as 1.5% which is very low and unsustainable. The usual protocol fees in similar protocols is something between 15% and 30% on accrued interest.

Description:
The protocol fees are too low and unsustainable compared to industry standards.

Recommended Mitigation:
Increase the protocol fee to a more sustainable level

Review: Acknowledged

[L-03] all the contracts are not upgradeable

Impact: LOW - No ability to fix bugs or add features after deployment

Description:
All contracts lack upgradeability mechanisms, limiting the ability to fix bugs or add features after deployment.

Recommended Mitigation:
Implement upgradeable patterns for critical contracts

Review: Acknowledged

[L-04] cancelOrders() spams the pool with empty orders

Impact: LOW - When canceling lending orders from the orderBook, the avonPool submits a list of empty orders instead of simply calling `OrderBook.batchInsertOrder()` with an empty list

Description:

```
function _cancelOrders(PoolStorage.PoolState storage s) internal {
    uint16 tickCount = _getTicks(s.config.oracle);
    (uint64[] memory rates, uint256[] memory liquidity) =
```

```
s.getQuoteSuggestions(
        tickCount,
        0
    );
    IOrderbook(s.orderBook).batchInsertOrder(
        rates,
        liquidity
    );
}
```

Recommended Mitigation:
Simply call `batchInsertOrder` with an empty list instead of generating empty orders.

Review: Fixed in `7d7a3648f0ab13759d2ea7236a06573f41ee4ef7`

[L-05] PoolManagers are not able to change the managerFee in AvonPool

Impact: LOW - It is expected for PoolManagers to be able to change the fee structure in order to make the pools under management more competitive or more profitable. E.g., when the pool is launched, usually the fees will be set to zero until considerable traction is achieved.

Description:
PoolManagers lack the ability to update fee structures, which limits their ability to make pools more competitive or profitable.

Recommended Mitigation:
We recommend adding a function to allow pool managers to update the fee structure.

Review: Acknowledged

[L-06] the `Vault` and the `AvonPool` don't implement any caps/limits on deposits/borrows

Impact: LOW - In order to manage risk in a vault/Pool it is highly recommended to add some limits/caps so that operations could be managed and the risk adjusted.

Description:
Both Vault and AvonPool contracts lack caps or limits on deposits and borrows, which could lead to unmanaged risk.

Recommended Mitigation:
We recommend implementing caps/checks on deposits/borrow in the pool and vault.

Review: Acknowledged

[L-07] PoolManagers are able to drain the pool

Impact: LOW - Centralization issue - a malicious pool manager is able to drain the pool, as they are allowed to change the orderBook to an address they control. They will then use this address to borrow on behalf of the protocol's users and thus drain the funds from the protocol.

Description:
PoolManagers have excessive control over critical parameters, creating centralization risks.

Recommended Mitigation:
Implement proper access controls and timelock mechanisms to prevent pool managers from changing critical parameters without oversight.

Review: Acknowledged

[L-08] missing stateUpdate for position.poolBorrowAssets,position.poolBorrowShares,position.updatedAt

Impact: LOW - Position state inconsistency during repay operations

Description:
Unlike liquidate or borrow operations, we are not updating the `position.poolBorrowAssets`, `position.poolBorrowShares`, and `position.updatedAt` while repaying.

Recommended Mitigation:
Update the position state variables during repay operations to maintain consistency with other operations.

Review: Fixed in `7d7a3648f0ab13759d2ea7236a06573f41ee4ef7`

[L-09] currentUtilization is not calculated correctly in `getQuoteSuggestions()`

Description:
In the case that the available liquidity is 0, the current utilization is not calculated correctly:

```
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 ? 0 :
(s.totalBorrowAssets * PoolConstants.MAX_UTILIZATION) /
s.totalSupplyAssets;
```

If the pool doesn't have any liquidity available, we will set the utilization rate to 0 (the opposite). For example, if a pool has 50k USDC as assets and the 50k USDC are borrowed out, the utilization should be 100 percent and not 0.

Recommended Mitigation:
To fix this, we should do the following:

```
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 ? 0 :
(s.totalBorrowAssets * PoolConstants.MAX_UTILIZATION) /
s.totalSupplyAssets;
+    uint256 currentUtilization = availableLiquidity == 0 ?
PoolConstants.MAX_UTILIZATION : (s.totalBorrowAssets *
PoolConstants.MAX_UTILIZATION) / s.totalSupplyAssets;
```

Review: Fixed in 9523c1bbd3bc96a396f8fe7491fa0baa03ecff82