SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Ajna
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: June 5 - June 22, 2023
Prepared on: July 25, 2023

Introduction

Ajna is a peer to peer, oracleless, permissionless lending protocol with no governance, accepting both fungible and non fungible tokens as collateral.

Scope

Repository: ajna-finance/ajna-core

Branch: main

Commit: e3632f6d0b196fb1bf1e59c05fb85daf357f2386

Repository: ajna-finance/ajna-grants

Branch: main

Commit: 65d52ce52039577b1cfefc76cbbf0030a87f4845

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
6	6

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

branch_indigo hyh Chinmay

Issue H-1: Pool's kickWithDeposit misses liquidation debt check

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/82

Found by

hyh

Summary

_revertIfAuctionDebtLocked() is missed in kickWithDeposit() function, that can actually remove deposit from anywhere, including HPB that are frozen by liquidation debt accumulator.

Vulnerability Detail

The inability to remove quote token deposit that is placed high enough to be covered by liquidation debt accumulator is a part of system design (see `7.5 Liquidation Debt` of Ajna protocol white paper).

The corresponding check is performed by _revertIfAuctionDebtLocked() in moveQuoteToken() and removeQuoteToken(), but is missed in kickWithDeposit() that allows for quote funds retrieval from HPB as well.

Impact

HPB depositors can use `kickWithDeposit() -> withdrawBonds()` for quote funds removal, effectively avoiding liquidation debt controls, which can lead to deposit shortage for the matters of eventual bad debt coverage. I.e. in some situations when depositor knows that his funds are about to be used to cover bad debt it might be reasonably for them to use kickWithDeposit() even knowing that there most probably will be a kicker penalty imposed.

This not only can create a number of bad faith auctions, but can move a burden of debt write offs to lower bucket depositors, who are unaware of such possibility and do not actively monitor pool state. This will allow HPB depositors to obtain stable yield, but off load a part of the corresponding risks, profiting off the lower buckets depositors (who, in general, pocketed a somewhat lower yield, but receive more risk this way).

As there is no low-probability prerequisites and the impact is a violation of system design allowing one group of users to profit off another, setting the severity to be high.

Code Snippet

kickWithDeposit() can effectively remove quote tokens from any bucket to cover kick bond, but is not controlled for liquidation debt buffer:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L321-L336

```
function kickWithDeposit(
    uint256 index_,
    uint256 npLimitIndex_
) external override nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

    // kick auctions
    KickResult memory result = KickerActions.kickWithDeposit(
        auctions,
        deposits,
        buckets,
        loans,
        poolState,
        index_,
        npLimitIndex_
    );
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L149-L243

```
    function kickWithDeposit(
        ...
    ) external returns (
        KickResult memory kickResult_
    ) {
        ...

        // kick top borrower
        kickResult_ = _kick(
            ...
        );

        // amount to remove from deposit covers entire bond amount
        if (vars.amountToDebitFromDeposit > kickResult_.amountToCoverBond) {
            // cap amount to remove from deposit at amount to cover bond
            vars.amountToDebitFromDeposit = kickResult_.amountToCoverBond;

            // recalculate the LUP with the amount to cover bond
            kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt +
↪   vars.amountToDebitFromDeposit);
```

```
            // entire bond is covered from deposit, no additional amount to be
↪   send by lender
            kickResult_.amountToCoverBond = 0;
        } else {
            // lender should send additional amount to cover bond
            kickResult_.amountToCoverBond -= vars.amountToDebitFromDeposit;
        }

        // revert if the bucket price used to kick and remove is below new LUP
        if (vars.bucketPrice < kickResult_.lup) revert PriceBelowLUP();

        // remove amount from deposits
        if (vars.amountToDebitFromDeposit == vars.bucketDeposit &&
↪   vars.bucketCollateral == 0) {
            // In this case we are redeeming the entire bucket exactly, and need
↪   to ensure bucket LP are set to 0
            vars.redeemedLP = vars.bucketLP;

>>          Deposits.unscaledRemove(deposits_, index_,
↪   vars.bucketUnscaledDeposit);
            vars.bucketUnscaledDeposit = 0;

        } else {
            ...

>>          Deposits.unscaledRemove(deposits_, index_, unscaledAmountToRemove);
            vars.bucketUnscaledDeposit -= unscaledAmountToRemove;
        }
```

_revertIfAuctionDebtLocked() is guarding direct quote funds removal via moveQuoteToken():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L176-L185

```
    function moveQuoteToken(
        uint256 maxAmount_,
        uint256 fromIndex_,
        uint256 toIndex_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 fromBucketLP_, uint256
↪   toBucketLP_, uint256 movedAmount_) {
        _revertAfterExpiry(expiry_);
        PoolState memory poolState = _accruePoolInterest();

>>      _revertIfAuctionDebtLocked(deposits, poolState.t0DebtInAuction,
↪   fromIndex_, poolState.inflator);
```

And removeQuoteToken():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L210-L218

```
    function removeQuoteToken(
        uint256 maxAmount_,
        uint256 index_
    ) external override nonReentrant returns (uint256 removedAmount_, uint256
↪   redeemedLP_) {
        _revertIfAuctionClearable(auctions, loans);

        PoolState memory poolState = _accruePoolInterest();

>>      _revertIfAuctionDebtLocked(deposits, poolState.t0DebtInAuction, index_,
↪   poolState.inflator);
```

Tool used

Manual Review

Recommendation

Consider adding the check:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L321-L336

```
    function kickWithDeposit(
        uint256 index_,
        uint256 npLimitIndex_
    ) external override nonReentrant {
        PoolState memory poolState = _accruePoolInterest();
+       _revertIfAuctionDebtLocked(deposits, poolState.t0DebtInAuction, index_,
↪   poolState.inflator);

        // kick auctions
        KickResult memory result = KickerActions.kickWithDeposit(
            auctions,
            deposits,
            buckets,
            loans,
            poolState,
            index_,
            npLimitIndex_
        );
```

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/894

dmitriia

> ajna-finance/contracts#894

Looks ok, `lenderKick()` (renamed `kickWithDeposit()`) no longer removes deposit, obtaining bond funds directly from the sender, so no liquidation debt check is now needed.

Issue H-2: kickWithDeposit removes the deposit without HTP pool state check

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/86

Found by

hyh

Summary

In order to cover kick bond KickerActions kickWithDeposit() removes the deposit from the pool, but misses the `new_LUP >= HTP` check, allowing for the invariant breaking state.

Vulnerability Detail

Every deposit removal in the protocol comes with the `LUP >= HTP` final state check, that ensures that active loans aren't eligible for liquidation (Ajna white paper `4.1 Deposit`).

kickWithDeposit() can effectively remove deposits, either partially or fully, but performs no such check, potentially leaving the pool in the `LUP < HTP` state.

Impact

A range of outcomes becomes possible after that, for example all other deposit operations can be frozen as long as they will not move LUP in the opposite direction, as their HTP checks will revert.

There is no low-probability prerequisites and the impact is a violation of the core system invariant, so setting the severity to be high.

Code Snippet

kickWithDeposit() can effectively remove quote tokens from any bucket to cover kick bond:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L321-L336

```
function kickWithDeposit(
    uint256 index_,
    uint256 npLimitIndex_
) external override nonReentrant {
    PoolState memory poolState = _accruePoolInterest();
```

```
    // kick auctions
    KickResult memory result = KickerActions.kickWithDeposit(
        auctions,
        deposits,
        buckets,
        loans,
        poolState,
        index_,
        npLimitIndex_
    );
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L149-L243

```
    function kickWithDeposit(
        ...
    ) external returns (
        KickResult memory kickResult_
    ) {
        ...

        // kick top borrower
        kickResult_ = _kick(
            ...
        );

        ...

        // remove amount from deposits
        if (vars.amountToDebitFromDeposit == vars.bucketDeposit &&
↪   vars.bucketCollateral == 0) {
            // In this case we are redeeming the entire bucket exactly, and need
↪   to ensure bucket LP are set to 0
            vars.redeemedLP = vars.bucketLP;

>>          Deposits.unscaledRemove(deposits_, index_,
↪   vars.bucketUnscaledDeposit);
            vars.bucketUnscaledDeposit = 0;

        } else {
            ...

>>          Deposits.unscaledRemove(deposits_, index_, unscaledAmountToRemove);
            vars.bucketUnscaledDeposit -= unscaledAmountToRemove;
        }
```

But there is no HTP check:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L242-L273

```
        vars.bucketUnscaledDeposit -= unscaledAmountToRemove;
    }

    vars.redeemedLP = Maths.min(vars.lenderLP, vars.redeemedLP);

    // revert if LP redeemed amount to kick auction is 0
    if (vars.redeemedLP == 0) revert InsufficientLP();

    uint256 bucketRemainingLP = vars.bucketLP - vars.redeemedLP;

    if (vars.bucketCollateral == 0 && vars.bucketUnscaledDeposit == 0 &&
↪   bucketRemainingLP != 0) {
        bucket.lps            = 0;
        bucket.bankruptcyTime = block.timestamp;

        emit BucketBankruptcy(
            ..
        );
    } else {
        // update lender and bucket LP balances
        lender.lps -= vars.redeemedLP;
        bucket.lps -= vars.redeemedLP;
    }

    emit RemoveQuoteToken(
        ...
    );
}
```

Tool used

Manual Review

Recommendation

Consider checking `LUP >= HTP` condition in the final state of the operation, similarly to other functions, for example removeQuoteToken():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L413-L424

```
lup_ = Deposits.getLup(deposits_, poolState_.debt);

uint256 htp = Maths.wmul(params_.thresholdPrice, poolState_.inflator);

if (
    // check loan book's htp doesn't exceed new lup
    htp > lup_
    ||
    // ensure that pool debt < deposits after removal
    // this can happen if lup and htp are less than min bucket price and htp >
↪   lup (since LUP is capped at min bucket price)
    (poolState_.debt != 0 && poolState_.debt > Deposits.treeSum(deposits_))
) revert LUPBelowHTP();
```

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/894

dmitriia

> ajna-finance/contracts#894

Looks ok, `lenderKick()` no longer removes deposit, obtaining bond funds directly from the sender, so HTP check now isn't needed.

Issue H-3: moveQuoteToken updates pool state using intermediary LUP, biasing pool's interest rate calculations

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/87

Found by

hyh

Summary

In LenderActions's moveQuoteToken() LUP is being evaluated after liquidity removal, but before liquidity addition. This intermediary LUP doesn't correspond to the final state of the pool, but is returned as if it does, leading to a bias in pool target utilization and interest rate calculations.

Vulnerability Detail

moveQuoteToken() calculates LUP after deposit removal only instead of doing so after the whole operation, being atomic removal from one index and addition to another, and then updates the pool accounting `_updateInterestState(poolState, newLup)` with this intermediary `newLup`, that doesn't correspond to the final state of the pool.

Impact

moveQuoteToken() is one of the base frequently used operations, so the state of the pool will be frequently enough updated with incorrect LUP and `EMA of LUP * t0 debt` internal accounting variable be systematically biased, which leads to incorrect interest rate dynamics of the pool.

There is no low-probability prerequisites and the impact is a bias in interest rate calculations, so setting the severity to be high.

Code Snippet

moveQuoteToken() calculates the LUP right after the deposit removal:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L237-L312

```
    function moveQuoteToken(
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
        PoolState calldata poolState_,
        MoveQuoteParams calldata params_
```

```
>>  ) external returns (uint256 fromBucketRedeemedLP_, uint256 toBucketLP_,
↪   uint256 movedAmount_, uint256 lup_) {
        ...

>>      (movedAmount_, fromBucketRedeemedLP_, vars.fromBucketRemainingDeposit) =
↪   _removeMaxDeposit(
            deposits_,
            RemoveDepositParams({
                depositConstraint: params_.maxAmountToMove,
                lpConstraint:      vars.fromBucketLenderLP,
                bucketLP:          vars.fromBucketLP,
                bucketCollateral:  vars.fromBucketCollateral,
                price:             vars.fromBucketPrice,
                index:             params_.fromIndex,
                dustLimit:         poolState_.quoteTokenScale
            })
        );

>>      lup_ = Deposits.getLup(deposits_, poolState_.debt);
        // apply unutilized deposit fee if quote token is moved from above the
↪   LUP to below the LUP
        if (vars.fromBucketPrice >= lup_ && vars.toBucketPrice < lup_) {
            movedAmount_ = Maths.wmul(movedAmount_, Maths.WAD -
↪   _depositFeeRate(poolState_.rate));
        }

        vars.toBucketUnscaledDeposit = Deposits.unscaledValueAt(deposits_,
↪   params_.toIndex);
        vars.toBucketScale           = Deposits.scale(deposits_,
↪   params_.toIndex);
        vars.toBucketDeposit         = Maths.wmul(vars.toBucketUnscaledDeposit,
↪   vars.toBucketScale);

        toBucketLP_ = Buckets.quoteTokensToLP(
            toBucket.collateral,
            toBucket.lps,
            vars.toBucketDeposit,
            movedAmount_,
            vars.toBucketPrice,
            Math.Rounding.Down
        );

        // revert if (due to rounding) the awarded LP in to bucket is 0
        if (toBucketLP_ == 0) revert InsufficientLP();

>>      Deposits.unscaledAdd(deposits_, params_.toIndex,
↪   Maths.wdiv(movedAmount_, vars.toBucketScale));
```


```
        vars.htp = Maths.wmul(params_.thresholdPrice, poolState_.inflator);

        // check loan book's htp against new lup, revert if move drives LUP
↪   below HTP
        if (params_.fromIndex < params_.toIndex && vars.htp > lup_) revert
↪   LUPBelowHTP();
```

Intermediary LUP is then being used for interest rate state update:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L176-L207

```
    function moveQuoteToken(
        uint256 maxAmount_,
        uint256 fromIndex_,
        uint256 toIndex_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 fromBucketLP_, uint256
↪   toBucketLP_, uint256 movedAmount_) {
        _revertAfterExpiry(expiry_);
        PoolState memory poolState = _accruePoolInterest();

        _revertIfAuctionDebtLocked(deposits, poolState.t0DebtInAuction,
↪   fromIndex_, poolState.inflator);

        uint256 newLup;
        (
            fromBucketLP_,
            toBucketLP_,
            movedAmount_,
>>          newLup
        ) = LenderActions.moveQuoteToken(
            buckets,
            deposits,
            poolState,
            MoveQuoteParams({
                maxAmountToMove: maxAmount_,
                fromIndex:       fromIndex_,
                toIndex:         toIndex_,
                thresholdPrice:  Loans.getMax(loans).thresholdPrice
            })
        );

        // update pool interest rate state
>>      _updateInterestState(poolState, newLup);
    }
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L675-L680

```
    function _updateInterestState(
        PoolState memory poolState_,
        uint256 lup_
    ) internal {

>>      PoolCommons.updateInterestState(interestState, emaState, deposits,
↪   poolState_, lup_);
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/PoolCommons.sol#L148-L152

```
// calculate the EMA of LUP * t0 debt
vars.luptODebtEma = uint256(
    PRBMathSD59x18.mul(vars.weightTu, int256(vars.luptODebtEma)) +
    PRBMathSD59x18.mul(1e18 - vars.weightTu, int256(interestParams_.luptODebt))
);
```

This will lead to a bias in target utilization and interest rate dynamics:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/PoolCommons.sol#L269-L289

```
function _calculateInterestRate(
    PoolState memory poolState_,
    uint256 debtEma_,
    uint256 depositEma_,
    uint256 debtColEma_,
    uint256 luptODebtEma_
) internal pure returns (uint256 newInterestRate_)  {
    // meaningful actual utilization
    int256 mau;
    // meaningful actual utilization * 1.02
    int256 mau102;

    if (poolState_.debt != 0) {
        // calculate meaningful actual utilization for interest rate update
        mau    = int256(_utilization(debtEma_, depositEma_));
        mau102 = mau * PERCENT_102 / 1e18;
    }

    // calculate target utilization
    int256 tu = (luptODebtEma_ != 0) ?
        int256(Maths.wdiv(debtColEma_, luptODebtEma_)) : int(Maths.WAD);
```

Tool used

Manual Review

Recommendation

Consider calculating LUP in moveQuoteToken() after deposit addition to the destination bucket. Deposit fee can be calculated from initial LUP only, so only one, final, LUP recalculation looks to be necessary.

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/891

dmitriia

> ajna-finance/contracts#891

Fix looks ok, final LUP is now used.

Issue H-4: Settlement can be called when auction period isn't concluded, allowing HPB depositors to game bad debt settlements

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/106

Found by

hyh

Summary

The end of auction period is included to it across the logic, but settlePoolDebt() treats the last moment as if it is beyond the period.

Vulnerability Detail

In settlePoolDebt() SettlerActions.sol#L113 the end of period control do not revert at `block.timestamp == kickTime + 72 hours`, allowing to run the settlement at the very last moment of the period.

Impact

Pool manipulations become possible at this point of time as both quote and collateral removal operations (guarded by `_revertIfAuctionClearable`) and settlePoolDebt() are available at this point of time.

As an example, HPB depositor can monitor pool state and upon the calculation that their bucket can be used for bad debt settlement, atomically run `removeQuoteToken() -> settlePoolDebt() -> addQuoteToken()` at `block.timestamp == kickTime + 72 hours`, retaining yield generating HPB position, while settling bad debt with funds of other depositors in nearby buckets.

While the probability looks to be medium, catching the exact moment is cumbersome, but achievable operation, the impact is one depositors profiting off others in a risk-free manner, so placing the overall severity to be medium.

Code Snippet

settlePoolDebt() can be run at `block.timestamp == kickTime + 72 hours`:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L100-L113

```
    function settlePoolDebt(
        AuctionsState storage auctions_,
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
        LoansState storage loans_,
        ReserveAuctionState storage reserveAuction_,
        PoolState calldata poolState_,
        SettleParams memory params_
    ) external returns (SettleResult memory result_) {
        uint256 kickTime = auctions_.liquidations[params_.borrower].kickTime;
        if (kickTime == 0) revert NoAuction();

        Borrower memory borrower = loans_.borrowers[params_.borrower];
>>      if ((block.timestamp - kickTime < 72 hours) && (borrower.collateral !=
 ↪  0)) revert AuctionNotClearable();
```

While `AuctionNotCleared()` is `block.timestamp - kickTime > 72 hours`, i.e. clearable auction is `[0, 72 hours]` period:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/helpers/RevertsHelper.sol#L50-L57

```
    function _revertIfAuctionClearable(
        AuctionsState storage auctions_,
        LoansState    storage loans_
    ) view {
        address head     = auctions_.head;
        uint256 kickTime = auctions_.liquidations[head].kickTime;
        if (kickTime != 0) {
>>          if (block.timestamp - kickTime > 72 hours) revert AuctionNotCleared();
```

Reserves take also includes the last timestamp to the period, proceeding with take:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L282-L291

```
    function takeReserves(
        ReserveAuctionState storage reserveAuction_,
        uint256 maxAmount_
    ) external returns (uint256 amount_, uint256 ajnaRequired_) {
        // revert if no amount to be taken
        if (maxAmount_ == 0) revert InvalidAmount();

        uint256 kicked = reserveAuction_.kicked;

>>      if (kicked != 0 && block.timestamp - kicked <= 72 hours) {
```

Tool used

Manual Review

Recommendation

Consider having settlePoolDebt() wait for the whole period to pass:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L100-L113

```
    function settlePoolDebt(
        ...
    ) external returns (SettleResult memory result_) {
        ...
-       if ((block.timestamp - kickTime < 72 hours) && (borrower.collateral !=
 ↪  0)) revert AuctionNotClearable();
+       if ((block.timestamp - kickTime <= 72 hours) && (borrower.collateral !=
 ↪  0)) revert AuctionNotClearable();
```

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/902

dmitriia

ajna-finance/contracts#902

Looks ok

Issue H-5: LUP is not recalculated after adding kicking penalty to pool's debt, so kick() updates the pool state with an outdated LUP

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/107

Found by

Chinmay, hyh

Summary

_kick() first calculates LUP, then adds kicking penalty, so the LUP returned without recalculation doesn't include the penalty and this way is outdated whenever it is not zero.

Vulnerability Detail

kick() and kickWithDeposit() (when deposit doesn't have any excess over the needed bond) returns _kick() calculated LUP, which is generally higher then real one being calculated before kicking penalty was added to the total debt.

Impact

kick() is one of the base frequently used operations, so the state of the pool will be frequently enough updated with incorrect LUP and `EMA of LUP * t0 debt` internal accounting variable be systematically biased, which leads to incorrect interest rate dynamics of the pool.

There is no low-probability prerequisites and the impact is a bias in interest rate calculations, so setting the severity to be high.

Code Snippet

kick() updates the `poolState` with _kick() returned `result.lup`:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L277-L313

```
    function kick(
        address borrower_,
        uint256 npLimitIndex_
    ) external override nonReentrant {
        PoolState memory poolState = _accruePoolInterest();
```

```
        // kick auction
>>      KickResult memory result = KickerActions.kick(
            auctions,
            deposits,
            loans,
            poolState,
            borrower_,
            npLimitIndex_
        );

        // update in memory pool state struct
        poolState.debt            =  Maths.wmul(result.t0PoolDebt,
↪   poolState.inflator);
        poolState.t0Debt          =  result.t0PoolDebt;
        poolState.t0DebtInAuction += result.t0KickedDebt;

        // adjust t0Debt2ToCollateral ratio
        _updateT0Debt2ToCollateral(
            result.debtPreAction,
            0, // debt post kick (for loan in auction) not taken into account
            result.collateralPreAction,
            0  // collateral post kick (for loan in auction) not taken into
↪   account
        );

        // update pool balances state
        poolBalances.t0Debt          = poolState.t0Debt;
        poolBalances.t0DebtInAuction = poolState.t0DebtInAuction;
        // update pool interest rate state
>>      _updateInterestState(poolState, result.lup);

        if (result.amountToCoverBond != 0) _transferQuoteTokenFrom(msg.sender,
↪   result.amountToCoverBond);
    }
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L115-L134

```
    function kick(
        ...
    ) external returns (
        KickResult memory
    ) {
>>      return _kick(
            auctions_,
            deposits_,
            loans_,
```

```
            poolState_,
            borrowerAddress_,
            limitIndex_,
            0
        );
    }
```

In _kick() kicking penalty is added to the total debt of the pool:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L438-L446

```
        // when loan is kicked, penalty of three months of interest is added
>>      vars.t0KickPenalty = Maths.wdiv(Maths.wmul(kickResult_.t0KickedDebt,
↪  poolState_.rate), 4 * 1e18);
        vars.kickPenalty   = Maths.wmul(vars.t0KickPenalty, poolState_.inflator);

>>      kickResult_.t0PoolDebt   = poolState_.t0Debt + vars.t0KickPenalty;
        kickResult_.t0KickedDebt += vars.t0KickPenalty;

        // update borrower debt with kicked debt penalty
        borrower.t0Debt = kickResult_.t0KickedDebt;
```

While the function calculates LUP before that (for _isCollateralized() check) and does not recalculate it after the penalty was added:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L364-L442

```
    function _kick(
        ...
    ) internal returns (
        KickResult memory kickResult_
    ) {
        ...
        // add amount to remove to pool debt in order to calculate proposed LUP
>>      kickResult_.lup          = Deposits.getLup(deposits_, poolState_.debt +
↪  additionalDebt_);

        ...

        // when loan is kicked, penalty of three months of interest is added
        vars.t0KickPenalty = Maths.wdiv(Maths.wmul(kickResult_.t0KickedDebt,
↪  poolState_.rate), 4 * 1e18);
        vars.kickPenalty   = Maths.wmul(vars.t0KickPenalty, poolState_.inflator);

>>      kickResult_.t0PoolDebt   = poolState_.t0Debt + vars.t0KickPenalty;
```

kickWithDeposit() returns _kick() calculated `kickResult_.lup` (i.e. before kick penalty) whenever `vars.amountToDebitFromDeposit <= kickResult_.amountToCoverBond`:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L190-L216

```
        // kick top borrower
>>      kickResult_ = _kick(
            auctions_,
            deposits_,
            loans_,
            poolState_,
            Loans.getMax(loans_).borrower,
            limitIndex_,
            vars.amountToDebitFromDeposit
        );

        // amount to remove from deposit covers entire bond amount
        if (vars.amountToDebitFromDeposit > kickResult_.amountToCoverBond) {
            // cap amount to remove from deposit at amount to cover bond
            vars.amountToDebitFromDeposit = kickResult_.amountToCoverBond;

            // recalculate the LUP with the amount to cover bond
            kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt +
 ↪  vars.amountToDebitFromDeposit);
            // entire bond is covered from deposit, no additional amount to be
 ↪  send by lender
            kickResult_.amountToCoverBond = 0;
>>      } else {
            // lender should send additional amount to cover bond
            kickResult_.amountToCoverBond -= vars.amountToDebitFromDeposit;
        }

        // revert if the bucket price used to kick and remove is below new LUP
        if (vars.bucketPrice < kickResult_.lup) revert PriceBelowLUP();
```

Tool used

Manual Review

Recommendation

Consider using initial LUP for _isCollateralized() check in _kick() as `additionalDebt_` is zero in plain kick() case, and calculating the final LUP at the end of _kick().

Consider refactoring kickWithDeposit(): for example, calculating the LUP therein with the corresponding `additionalDebt_` after _kick() call, and adding the flag to _kick() call to indicate that LUP calculation isn't needed.

Discussion

grandizzy

part of PR https://github.com/ajna-finance/contracts/pull/894 that change `kickWithDeposit` functionality https://github.com/ajna-finance/contracts/pull/894/files#diff-54056532b4b7aac8940fbec13725e98ceceb358bef02e1285edad2741dff83bdR363

dmitriia

> part of PR ajna-finance/contracts#894 that change `kickWithDeposit` functionality https://github.com/ajna-finance/contracts/pull/894/files#diff-54056532b4b7aac8940fbec13725e98ceceb358bef02e1285edad2741dff83bdR363

Fix looks good, `_kick()` is the last operation in `kick()` and `lenderKick()` (which is the new version of `kickWithDeposit()`), and LUP is calculated in `_kick()` after all the changes off final structures.

Issue H-6: Debt write off can be prohibited by HPB depositor by continuously allocating settlement blocking dust deposits in the higher buckets

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/110

Found by

hyh

Summary

High price bucket depositor who faces bad debt settlement can add multiple dust quote token deposits to many higher price buckets and stale settlement.

Vulnerability Detail

HPB depositor have incentives to and actually can defend themselves from using their deposits in bad debt write offs by doing multiple dust quote token deposits in vast number of higher price buckets (up to and above current market price). This will stale bad debt settlement: now logic only requires amount to be positive, SettlerActions.sol#L334-L356, and it is possible to add quote token dust, Pool.sol#L146-L166, LenderActions.sol#L148-L157.

The point in doing so is that, having the deposit frozen is better then participate in a write off, which is a direct loss, as:

1) other unaware depositors might come in and free the HPB depositor from liquidation debt participation, possibly taking bad debt damage,
2) the HPB depositor can still bucketTake() as there is no _revertIfAuctionDebtLocked() check. As it will generate collateral instead of quote funds, it might be then retrieved by removeCollateral().

When there is low amount of debt in liquidation, removing this dust deposits is possible, but economically not feasible: despite high price used gas cost far exceeds the profit due to quote amount being too low.

When there is substantial amount of debt in liquidation, direct removal via removeQuoteToken() will be blocked by _revertIfAuctionDebtLocked() control, while `settle() -> settlePoolDebt()` calls will be prohibitively expensive (will go trough all the dust populated buckets) and fruitless (only dust amount will be settled), while the defending HPB depositor can simultaneously add those dust deposits back.

Economically the key point here is that incentives of the defending HPB depositor are more substantial (they will suffer principal loss on bad debt settlement) than the incentives of agents who call `settle() -> settlePoolDebt()` (they have their

lower bucket deposits temporary frozen and want to free them with settling bad debt with HPB deposit).

Impact

HPB depositors can effectively avoid deposit write off for bad debt settlement. I.e. in some situations when HPB depositor is a whale closely monitoring the pool and knowing that his funds are about to be used to cover a substantial amount of bad debt, the cumulative gas costs of the described strategy will be far lower than the gain of having principal funds recovered over time via `takeBucket() -> removeCollateral()`.

This will cause bad debt to pile up and stale greater share of the pool. The HPB depositor will eventually profit off from other depositors, who do not actively monitor pool state and over time participate in the bad debt settlements by placing deposits among the dust ones. This will allow the HPB depositor to obtain stable yield all this time, but off load a part of the corresponding risks.

As there is no low-probability prerequisites and the impact is a violation of system design allowing one group of users to profit off another, setting the severity to be high.

Code Snippet

There is no dust control in addQuoteToken():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L146-L166

```
function addQuoteToken(
    uint256 amount_,
    uint256 index_,
    uint256 expiry_
) external override nonReentrant returns (uint256 bucketLP_) {
    _revertAfterExpiry(expiry_);
    PoolState memory poolState = _accruePoolInterest();

    // round to token precision
    amount_ = _roundToScale(amount_, poolState.quoteTokenScale);

    uint256 newLup;
    (bucketLP_, newLup) = LenderActions.addQuoteToken(
        buckets,
        deposits,
        poolState,
        AddQuoteParams({
            amount: amount_,
            index:  index_
```

```
        })
    );
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L148-L157

```
function addQuoteToken(
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    PoolState calldata poolState_,
    AddQuoteParams calldata params_
) external returns (uint256 bucketLP_, uint256 lup_) {
    // revert if no amount to be added
    if (params_.amount == 0) revert InvalidAmount();
    // revert if adding to an invalid index
    if (params_.index == 0 || params_.index > MAX_FENWICK_INDEX) revert
↪   InvalidIndex();
```

Putting dust in lots of higher buckets will freeze the settlement as there no control over amount to operate with on every iteration, while `bucketDepth_` is limited and there is a block gas limit:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L334-L356

```
    function _settlePoolDebtWithDeposit(
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
        SettleParams memory params_,
        Borrower memory borrower_,
        uint256 inflator_
    ) internal returns (uint256 remainingt0Debt_, uint256 remainingCollateral_,
↪   uint256 bucketDepth_) {
        remainingt0Debt_     = borrower_.t0Debt;
        remainingCollateral_ = borrower_.collateral;
        bucketDepth_         = params_.bucketDepth;

        while (bucketDepth_ != 0 && remainingt0Debt_ != 0 &&
↪   remainingCollateral_ != 0) {
            SettleLocalVars memory vars;

>>          (vars.index, , vars.scale) =
↪   Deposits.findIndexAndSumOfSum(deposits_, 1);
            vars.hpbUnscaledDeposit    = Deposits.unscaledValueAt(deposits_,
↪   vars.index);
            vars.unscaledDeposit       = vars.hpbUnscaledDeposit;
            vars.price                 = _priceAt(vars.index);
```

```
>>          if (vars.unscaledDeposit != 0) {
                vars.debt              = Maths.wmul(remainingtODebt_,
↪   inflator_);           // current debt to be settled
                vars.maxSettleableDebt = Maths.floorWmul(remainingCollateral_,
↪   vars.price); // max debt that can be settled with existing collateral
                vars.scaledDeposit     = Maths.wmul(vars.scale,
↪   vars.unscaledDeposit);
```

The owner of such deposit can still use it for bucketTake() as there is no _revertIfAuctionDebtLocked() check there (which is ok by itself as the operation reduces the liquidation debt):

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L133-L164

```
    function bucketTake(
        ...
    ) external returns (TakeResult memory result_) {
        Borrower memory borrower = loans_.borrowers[borrowerAddress_];
        // revert if borrower's collateral is 0
        if (borrower.collateral == 0) revert InsufficientCollateral();

        result_.debtPreAction       = borrower.tODebt;
        result_.collateralPreAction = borrower.collateral;

        // bucket take auction
>>      TakeLocalVars memory vars = _takeBucket(
            auctions_,
            buckets_,
            deposits_,
            borrower,
            BucketTakeParams({
                borrower:        borrowerAddress_,
                inflator:        poolState_.inflator,
                depositTake:     depositTake_,
                index:           index_,
                collateralScale: collateralScale_
            })
        );
```

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L415-L464

```
function _takeBucket(
    ...
) internal returns (TakeLocalVars memory vars_) {
```

```
    ...
    _rewardBucketTake(
        auctions_,
        deposits_,
        buckets_,
        liquidation,
        params_.index,
        params_.depositTake,
        vars_
    );
```

During _rewardBucketTake() the principal quote funds are effectively exchanged with the collateral:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L606-L667

```
    function _rewardBucketTake(
        ...
    ) internal {

        ...

        // remove quote tokens from buckets deposit
>>      Deposits.unscaledRemove(deposits_, bucketIndex_,
↪   vars.unscaledQuoteTokenAmount);

        // total rewarded LP are added to the bucket LP balance
        if (totalLPReward != 0) bucket.lps += totalLPReward;
        // collateral is added to the buckets claimable collateral
>>      bucket.collateral += vars.collateralAmount;
```

So the HPB depositor can remove it (there is no _revertIfAuctionDebtLocked() check for collateral):

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/ERC20Pool.sol#L318-L335

```
    function removeCollateral(
        uint256 maxAmount_,
        uint256 index_
    ) external override nonReentrant returns (uint256 removedAmount_, uint256
↪   redeemedLP_) {
        _revertIfAuctionClearable(auctions, loans);

        PoolState memory poolState = _accruePoolInterest();

        // round the collateral amount appropriately based on token precision
```

```
        maxAmount_ = _roundToScale(maxAmount_, _getArgUint256(COLLATERAL_SCALE));

>>      (removedAmount_, redeemedLP_) = LenderActions.removeMaxCollateral(
            buckets,
            deposits,
            _bucketCollateralDust(index_),
            maxAmount_,
            index_
        );
```

But this means that there is no downside in doing so, but it is a significant upside in effectively denying the bad debt settlements.

I.e. the HPB depositor will place his deposit high, gain yield, and when his bucket happens to be within liquidation debt place these dust deposits to prevent settlements. Their deposit will be exchangeable to collateral on bucketTake() over a while, and it's still far better situation than taking part in debt write-off.

Tool used

Manual Review

Recommendation

There might be different design approaches to limiting such a strategy. As an example, consider controlling addQuoteToken() for dust (the limit might be a pool parameter set on deployment with the corresponding explanations that it shouldn't be loo low) and/or controlling it for deposit addition to be buckets higher than current HPB when there is a liquidation debt present (this will also shield naive depositors as such deposits can be subject to write offs, which they can be unaware of, i.e. the reward-risk of such action doesn't look good, so it can be prohibited for both reasons).

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/909

dmitriia

As far as I see there still is a surface of performing such bad debt write off protection just before auction becomes clearable, say by front-running the tx that changes the state so it becomes clearable (say tx removes the last piece of collateral and `borrower.t0Debt != 0 && borrower.collateral == 0` becomes true). I.e. HPB depositor might monitor the auction and run multiple `addQuoteToken()` just

before `_revertIfAuctionClearable()` starts to trigger for a big chunk of bad debt now auctioned.

Also, why can't attacker preliminary open another 'protection' deposit far from the top (to avoid liquidation debt, as its size can be small it doesn't have to be yield bearing) and use `moveQuoteToken()` to populate higher buckets with dust as described in the issue?

For this end the similar logic can be added to `moveQuoteToken()`, e.g.:

https://github.com/ajna-finance/contracts/blob/04adfedd597ba19fd362f82d14a85a1649d87ebf/src/base/Pool.sol#L181-L187

```
    function moveQuoteToken(
        uint256 maxAmount_,
        uint256 fromIndex_,
        uint256 toIndex_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 fromBucketLP_, uint256
 ↪  toBucketLP_, uint256 movedAmount_) {
        _revertAfterExpiry(expiry_);

+       _revertIfAuctionClearable(auctions, loans);
```

grandizzy

> Also, why can't attacker preliminary open another 'protection' deposit far from the top (to avoid liquidation debt, as its size can be small it doesn't have to be yield bearing) and use `moveQuoteToken()` to populate higher buckets with dust as described in the issue?
>
> For this end the similar logic can be added to `moveQuoteToken()`, e.g.:
>
> https://github.com/ajna-finance/contracts/blob/04adfedd597ba19fd362f82d14a85a1649d87ebf/src/base/Pool.sol#L181-L187
>
> ```
>     function moveQuoteToken(
>         uint256 maxAmount_,
>         uint256 fromIndex_,
>         uint256 toIndex_,
>         uint256 expiry_
>     ) external override nonReentrant returns (uint256 fromBucketLP_,
>  ↪  uint256 toBucketLP_, uint256 movedAmount_) {
>         _revertAfterExpiry(expiry_);
> 
> +       _revertIfAuctionClearable(auctions, loans);
> ```

PR to add same for move quote token
https://github.com/ajna-finance/contracts/pull/919

dmitriia

To prevent the attack during ongoing auction, e.g. in the front running manner as just described, there is a good option of prohibiting the high price deposits on addition and moving:

https://github.com/ajna-finance/contracts/blob/04adfedd597ba19fd362f82d14a85a1649d87ebf/src/base/Pool.sol#L146-L158

```
    function addQuoteToken(
        uint256 amount_,
        uint256 index_,
        uint256 expiry_,
        bool    revertIfBelowLup_
    ) external override nonReentrant returns (uint256 bucketLP_) {
        _revertAfterExpiry(expiry_);

        _revertIfAuctionClearable(auctions, loans);

        PoolState memory poolState = _accruePoolInterest();

+       _revertIfAboveHeadAuctionPrice(..., index_);
```

https://github.com/ajna-finance/contracts/blob/04adfedd597ba19fd362f82d14a85a1649d87ebf/src/base/Pool.sol#L180-L191

```
    /// @inheritdoc IPoolLenderActions
    function moveQuoteToken(
        uint256 maxAmount_,
        uint256 fromIndex_,
        uint256 toIndex_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 fromBucketLP_, uint256
 ↪  toBucketLP_, uint256 movedAmount_) {
        _revertAfterExpiry(expiry_);
        PoolState memory poolState = _accruePoolInterest();

        _revertIfAuctionDebtLocked(deposits, poolState.t0DebtInAuction,
 ↪  fromIndex_, poolState.inflator);

+       _revertIfAboveHeadAuctionPrice(..., toIndex_);
```

`_revertIfAboveHeadAuctionPrice()` is a new control function:

https://github.com/ajna-finance/contracts/blob/94be5c24dc448a9a0e914036450ac57b00c5ad11/src/libraries/helpers/RevertsHelper.sol#L50-L62

```
function _revertIfAboveHeadAuctionPrice(
```

```
    ...
    uint256 index_
) view {
    address head     = auctions_.head;
    uint256 kickTime = auctions_.liquidations[head].kickTime;
    ...
    if (_auctionPrice(momp, NP, kickTime) < _priceAt(index_)) revert
↪   PriceTooHigh();
}
```

As adding deposits above auction price is generally harmful for depositors (they will be a subject to immediate arbitrage), this will also shield against such uninformed user behavior.

dmitriia

> PR to add same for move quote token ajna-finance/contracts#919

Looks ok

Issue M-1: Lenders lose interests and pay deposit fees due to no slippage control

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/72

Found by

branch_indigo

Summary

When a lender deposits quote tokens below the minimum of LUP(Lowest Utilization Price) and HTP(Highest Threshold Price), the deposits will not earn interest and will also be charged deposit fees, according to docs. When a lender deposits to a bucket, they are vulnerable to pool LUP slippage which might cause them to lose funds due to fee charges against their will.

Vulnerability Detail

A lender would call `addQuoteToken()` to deposit. This function only allows entering expiration time for transaction settlement, but there is no slippage protection.

```
//Pool.sol
    function addQuoteToken(
        uint256 amount_,
        uint256 index_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 bucketLP_) {
        _revertAfterExpiry(expiry_);
        PoolState memory poolState = _accruePoolInterest();
        // round to token precision
        amount_ = _roundToScale(amount_, poolState.quoteTokenScale);
        uint256 newLup;
        (bucketLP_, newLup) = LenderActions.addQuoteToken(
            buckets,
            deposits,
            poolState,
            AddQuoteParams({
                amount: amount_,
                index:  index_
            })
        );
       ...
```

In LenderActions.sol, `addQuoteToken()` takes current `DepositsState` in storage and

current `poolState_.debt` in storage to calculate spot LUP prior to deposit. And this LUP is compared with user input bucket `index_` to determine if the lender will be punished with deposit fees. The deposit amount is then written to storage.

```
//LenderActions.sol
    function addQuoteToken(
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
        PoolState calldata poolState_,
        AddQuoteParams calldata params_
    ) external returns (uint256 bucketLP_, uint256 lup_) {
  ...
          // charge unutilized deposit fee where appropriate
 |>       uint256 lupIndex = Deposits.findIndexOfSum(deposits_, poolState_.debt);
        bool depositBelowLup = lupIndex != 0 && params_.index > lupIndex;
        if (depositBelowLup) {
            addedAmount = Maths.wmul(addedAmount, Maths.WAD -
↪   _depositFeeRate(poolState_.rate));
        }
...
   Deposits.unscaledAdd(deposits_, params_.index, unscaledAmount);
...
```

It should be noted that current `deposits_` and `poolState_.debt` can be different from when the user invoked the transaction, which will result in a different LUP spot price unforeseen by the lender to determine deposit fees. Even though lenders can input a reasonable expiration time `expirty_`, this will only prevent stale transactions to be executed and not offer any slippage control.

When there are many lenders depositing around the same time, LUP spot price can be increased and if the user transaction settles after a whale lender which moves the LUP spot price up significantly, the user might get accidentally punished for depositing below LUP. Or there could also be malicious lenders trying to ensure their transactions settle at a favorable LUP/HTP and front-run the user transaction, in which case the user transaction might still settle after the malicious lender and potentially get charged for fees.

Impact

Lenders might get charged deposit fees due to slippage against their will with or without MEV attacks, lenders might also lose on interest by depositing below HTP.

Code Snippet

https://github.com/ajna-finance/ajna-core/blob/e3632f6d0b196fb1bf1e59c05fb85daf357f2386/src/base/Pool.sol#L146-L150

https://github.com/ajna-finance/ajna-core/blob/e3632f6d0b196fb1bf1e59c05fb85daf357f2386/src/libraries/external/LenderActions.sol

https://github.com/ajna-finance/ajna-core/blob/e3632f6d0b196fb1bf1e59c05fb85daf357f2386/src/libraries/external/LenderActions.sol#L195

Tool used

Manual Review

Recommendation

Add slippage protection in Pool.sol `addQuoteToken()`. A lender can enable slippage protection, which will enable comparing deposit `index_` with `lupIndex` in LenderActions.sol.

Discussion

ith-harvey

We think this should be a low. Although not explicitly stated that this can happen in docs it is assumed based off of implementation. We were aware, not concerned.

grandizzy

https://github.com/ajna-finance/contracts/pull/915

0xffff11

I can still see the issue as a medium. Sponsor agreed to the issue and it has some impact on the fees that lender might have to pay

dmitriia

> ajna-finance/contracts#915

Looks ok, but shouldn't the same flag be introduced to `moveQuoteToken()`, e.g.:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/LenderActions.sol#L290-L292

```
    if (vars.fromBucketPrice >= lup_ && vars.toBucketPrice < lup_) {
+       if (params_.revertIfBelowLup) revert PriceBelowLUP();
        movedAmount_ = Maths.wmul(movedAmount_, Maths.WAD -
↪   _depositFeeRate(poolState_.rate));
    }
```

grandizzy

> ajna-finance/contracts#915

> Looks ok, but shouldn't the same flag be introduced to `moveQuoteToken()`, e.g.:
>
> https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/LenderActions.sol#L290-L292

```
    if (vars.fromBucketPrice >= lup_ && vars.toBucketPrice < lup_) {
+       if (params_.revertIfBelowLup) revert PriceBelowLUP();
        movedAmount_ = Maths.wmul(movedAmount_, Maths.WAD -
↪   _depositFeeRate(poolState_.rate));
    }
```

implemented with https://github.com/ajna-finance/contracts/pull/918 Beside suggested change there are 2 additional updates

- `MoveQuoteParams` struct moved from inline in order to avoid stack too deep error
- shrink `Pool` contract size by reading structs in view functions only once

dmitriia

> implemented with ajna-finance/contracts#918

Looks ok, the above logic now added.

ctf-sec

Escalate for 10 USDC.

As the sponsor said

> We think this should be a low. Although not explicitly stated that this can happen in docs it is assumed based off of implementation. We were aware, not concerned.

this is more like a feature request, not bug

sherlock-admin

> Escalate for 10 USDC.
>
> As the sponsor said
>
> > We think this should be a low. Although not explicitly stated that this can happen in docs it is assumed based off of implementation. We were aware, not concerned.
>
> this is more like a feature request, not bug

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

bzpassersby

I think this issue is a valid medium. It points to a possible scenario where a lender has to pay fees and lose interest against their will due to no slippage protection, which can be exploited through MEV attack. Because of the scenario of lenders losing funds against their intention, it should be medium.

Due to the fact that slippage can be exploited to cause users to lose funds, this should be considered a vulnerability or bug.

0xffff11

I disagree with the escalation in this case. I think it should be a medium. Despite being a design decision, allows users to be exposed to high slippage on behalf of their decision.

dmitriia

Looks like valid medium to me, the probability of the material impact can be said to be medium, so is the impact itself.

MLON33

> implemented with ajna-finance/contracts#918
>
> Looks ok, the above logic now added.

Moving sign-off by @dmitriia here for clarity.

hrishibhat

Result: Medium Has duplicates Considering a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ctf-sec: rejected

Issue M-2: Due to excessive HTP check moveQuoteToken can be unavailable for big deposits

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/84

Found by

hyh

Summary

moveQuoteToken() will revert if deposit removal causes LUP to be less than HTP, while the whole operation being the *removal and addition* to another index, so the check structured this way is excessive and prohibit a substantial share of active debt management operations. I.e. it has to be `HTP <= LUP` before and after the move, not in-between.

Vulnerability Detail

The one of the main purposes of moveQuoteToken() is to allow for dynamic management of deposit placement within the pool. This is crucial for controlling the associated risks: quote funds within the buckets can be traded with collateral at bucket's price, high price buckets can be frozen for liquidation debt buffer, then they can take part in debt write off. On the other hand low price buckets will not receive any yield while their price is below HTP (Ajna white paper `4.1 Deposit` and others). This way for any depositor the ability to move the quote funds between buckets is crucial.

The unavailability of moveQuoteToken() due to excess `HTP > LUP` check can directly lead to losses for the corresponding depositor. I.e. they can place funds to the higher price bucket, expecting to manage them closely to mitigate risks, so they can monitor the situation, being ready to move the funds out immediately when situation worsens (collateral market price moves down, liquidation volume spikes, and so on), but find themselves unable to do so.

Impact

moveQuoteToken() can be frequently unavailable (especially in big deposits case), which can directly lead to the depositor's losses.

Probability of unavailability looks to be high (removal of big enough deposit can frequently cause `HTP > LUP` state), while the probability of the following loss is medium, so placing the severity to medium as well.

Code Snippet

moveQuoteToken() is for moving deposit from `fromIndex_` to `toIndex_`:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/Pool.sol#L176-L207

```
    function moveQuoteToken(
        uint256 maxAmount_,
        uint256 fromIndex_,
        uint256 toIndex_,
        uint256 expiry_
    ) external override nonReentrant returns (uint256 fromBucketLP_, uint256
↪   toBucketLP_, uint256 movedAmount_) {
        ...

        (
            fromBucketLP_,
            toBucketLP_,
            movedAmount_,
            newLup
>>      ) = LenderActions.moveQuoteToken(
            buckets,
            deposits,
            poolState,
            MoveQuoteParams({
                maxAmountToMove: maxAmount_,
                fromIndex:       fromIndex_,
                toIndex:         toIndex_,
                thresholdPrice:  Loans.getMax(loans).thresholdPrice
            })
        );
        ...
    }
```

moveQuoteToken() controls for `vars.htp > lup_` with LUP being calculated after deposit removal, but before adding it back to the destination bucket:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L237-L312

```
    function moveQuoteToken(
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
        PoolState calldata poolState_,
        MoveQuoteParams calldata params_
    ) external returns (uint256 fromBucketRedeemedLP_, uint256 toBucketLP_,
↪   uint256 movedAmount_, uint256 lup_) {
```

```
        ...

>>      (movedAmount_, fromBucketRedeemedLP_, vars.fromBucketRemainingDeposit) =
↪   _removeMaxDeposit(
            deposits_,
            RemoveDepositParams({
                depositConstraint: params_.maxAmountToMove,
                lpConstraint:      vars.fromBucketLenderLP,
                bucketLP:          vars.fromBucketLP,
                bucketCollateral:  vars.fromBucketCollateral,
                price:             vars.fromBucketPrice,
                index:             params_.fromIndex,
                dustLimit:         poolState_.quoteTokenScale
            })
        );

>>      lup_ = Deposits.getLup(deposits_, poolState_.debt);
        // apply unutilized deposit fee if quote token is moved from above the
↪   LUP to below the LUP
        if (vars.fromBucketPrice >= lup_ && vars.toBucketPrice < lup_) {
            movedAmount_ = Maths.wmul(movedAmount_, Maths.WAD -
↪   _depositFeeRate(poolState_.rate));
        }

        vars.toBucketUnscaledDeposit = Deposits.unscaledValueAt(deposits_,
↪   params_.toIndex);
        vars.toBucketScale           = Deposits.scale(deposits_,
↪   params_.toIndex);
        vars.toBucketDeposit         = Maths.wmul(vars.toBucketUnscaledDeposit,
↪   vars.toBucketScale);

        toBucketLP_ = Buckets.quoteTokensToLP(
            toBucket.collateral,
            toBucket.lps,
            vars.toBucketDeposit,
            movedAmount_,
            vars.toBucketPrice,
            Math.Rounding.Down
        );

        // revert if (due to rounding) the awarded LP in to bucket is 0
        if (toBucketLP_ == 0) revert InsufficientLP();

>>      Deposits.unscaledAdd(deposits_, params_.toIndex,
↪   Maths.wdiv(movedAmount_, vars.toBucketScale));

        vars.htp = Maths.wmul(params_.thresholdPrice, poolState_.inflator);
```

```
        // check loan book's htp against new lup, revert if move drives LUP
↪  below HTP
>>      if (params_.fromIndex < params_.toIndex && vars.htp > lup_) revert
↪  LUPBelowHTP();
```

Tool used

Manual Review

Recommendation

Consider controlling `params_.fromIndex < params_.toIndex && vars.htp > lup_` with `lup_` being the final LUP, calculated after `Deposits.unscaledAdd(deposits_, params_.toIndex, Maths.wdiv(movedAmount_, vars.toBucketScale))`.

Deposit fee can be calculated from initial LUP only: it looks that if deposit fee condition is true for initial LUP then LUP will not change, if it is true for final LUP then it wasn't changed.

Discussion

grandizzy

fix in https://github.com/ajna-finance/contracts/pull/891

dmitriia

> fix in ajna-finance/contracts#891

Looks ok, LUP is being recalculated:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/LenderActions.sol#L310-L313

```
Deposits.unscaledAdd(deposits_, params_.toIndex, Maths.wdiv(movedAmount_,
↪  vars.toBucketScale));

// recalculate LUP after adding amount in to bucket only if to bucket price is
↪  greater than LUP
if (vars.toBucketPrice > lup_) lup_ = Deposits.getLup(deposits_,
↪  poolState_.debt);
```

Issue M-3: Limit index isn't checked in repayDebt, so user control is void

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/85

Found by

hyh

Summary

repayDebt() resulting LUP `_revertIfPriceDroppedBelowLimit()` check is not performed in the case of pure debt repayment without collateral pulling.

Vulnerability Detail

LUP will move (up or no change) as a result of debt repayment and repayDebt() have `limitIndex_` argument. As a part of multi-position strategy a user might not be satisfied with repay results if LUP has increased not substantially enough.

I.e. there is a user control argument, it is detrimental from UX perspective to request, but not use it, as for any reason a borrower might want to control for that move: they might expect the final level to be somewhere, as an example for the sake of other loans of that borrower.

Impact

Unfavorable repayDebt() operations will be executed and the borrowers, whose strategies were dependent on the realized LUP move, can suffer a loss.

Probability of execution is high (no prerequisites, current ordinary behavior), while the probability of the following loss is medium, so placing the severity to be medium.

Code Snippet

There is a `limitIndex_` parameter in repayDebt():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/ERC20Pool.sol#L208-L232

```
    function repayDebt(
        address borrowerAddress_,
        uint256 maxQuoteTokenAmountToRepay_,
        uint256 collateralAmountToPull_,
        address collateralReceiver_,
```

```
>>      uint256 limitIndex_
    ) external nonReentrant {
        ...

        RepayDebtResult memory result = BorrowerActions.repayDebt(
            auctions,
            buckets,
            deposits,
            loans,
            poolState,
            borrowerAddress_,
            maxQuoteTokenAmountToRepay_,
            collateralAmountToPull_,
>>          limitIndex_
        );
```

Currently `_revertIfPriceDroppedBelowLimit()` is done on collateral pulling only:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/BorrowerActions.sol#L365-L375

```
        if (vars.pull) {
            // only intended recipient can pull collateral
            if (borrowerAddress_ != msg.sender) revert BorrowerNotSender();

            // an auctioned borrower in not allowed to pull collateral (even if
↪   collateralized at the new LUP) if auction is not settled
            if (result_.inAuction) revert AuctionActive();

            // calculate LUP only if it wasn't calculated in repay action
            if (!vars.repay) result_.newLup = Deposits.getLup(deposits_,
↪   result_.poolDebt);

>>          _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
```

Tool used

Manual Review

Recommendation

Consider adding the same check in the repayment part:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/BorrowerActions.sol#L328

```
            result_.newLup = Deposits.getLup(deposits_, result_.poolDebt);
+           _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
```

If no repay or pull it looks ok to skip the check to save gas.

Discussion

grandizzy

Repayment can only make the position less risky, so don't see a need for frontrunning/stale TX protection. Documentation should clearly state this behavior.

0xffff11

I do see the issue being valid. As sponsor said, they don't see a need to implement a safeguard, but the watson demonstrated that users can suffer a loss if non-favorable repays are executed. Could keep the medium.

grandizzy

re discussed within team and we're going to provide a change for this behavior: https://github.com/ajna-finance/contracts/pull/914

0xffff11

Will keep the medium severity

dmitriia

> re discussed within team and we're going to provide a change for this behavior: ajna-finance/contracts#914

Looks ok, `limitIndex_` in `repayDebt()` is now effective in all the cases.

But it looks like when `vars.repay == vars.pull == false` the check will always revert the call as `newPrice_ = result_.newLup == 0`:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/helpers/RevertsHelper.sol#L71-L76

```
function _revertIfPriceDroppedBelowLimit(
    uint256 newPrice_,
    uint256 limitIndex_
) pure {
    if (newPrice_ < _priceAt(limitIndex_)) revert LimitIndexExceeded();
}
```

Consider:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/BorrowerActions.sol#L393-L394

```
        // check limit price and revert if price dropped below
-       _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
+       if (vars.pull || vars.repay)
 ↪  _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
```

grandizzy

> re discussed within team and we're going to provide a change for this behavior: ajna-finance/contracts#914

Looks ok, `limitIndex_` in `repayDebt()` is now effective in all the cases.

But it looks like when `vars.repay == vars.pull == false` the check will always revert the call as `newPrice_ = result_.newLup == 0`:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/helpers/RevertsHelper.sol#L71-L76

```
function _revertIfPriceDroppedBelowLimit(
    uint256 newPrice_,
    uint256 limitIndex_
) pure {
    if (newPrice_ < _priceAt(limitIndex_)) revert LimitIndexExceeded();
}
```

Consider:

https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/BorrowerActions.sol#L393-L394

```
        // check limit price and revert if price dropped below
-       _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
+       if (vars.pull || vars.repay)
 ↪  _revertIfPriceDroppedBelowLimit(result_.newLup, limitIndex_);
```

in that case we early revert with InvalidAmount at L288
https://github.com/ajna-finance/contracts/blob/0332f341856e1efe4da8bb675886c8cfbee57b71/src/libraries/external/BorrowerActions.sol#L288

```
// revert if no amount to pull or repay
if (!vars.repay && !vars.pull) revert InvalidAmount();
```

so later check not needed

dmitriia

Looks ok

Issue M-4: LenderActions's moveQuoteToken can create a total debt undercoverage

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/88

Found by

hyh

Summary

moveQuoteToken() doesn't ensure that pool debt is less than deposits after operation, while unutilized deposit fee can reduce total deposits as a result of the move.

Vulnerability Detail

Unutilized deposit fee can create a `poolState_.debt > Deposits.treeSum(deposits_)` state, which isn't controlled for in moveQuoteToken().

Impact

Pool can enter technical corner case when LUP is actually lower than HTP, numerically it will not be the case due to bounded nature of LUP calculation.

This breaks the core logic of the pool with the corresponding material miscalculations, but has low probability, so setting the severity to be medium.

Code Snippet

moveQuoteToken() can reduce overall deposits due to unutilized deposit fee incurred:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L285-L289

```
lup_ = Deposits.getLup(deposits_, poolState_.debt);
// apply unutilized deposit fee if quote token is moved from above the LUP to
↪  below the LUP
if (vars.fromBucketPrice >= lup_ && vars.toBucketPrice < lup_) {
    movedAmount_ = Maths.wmul(movedAmount_, Maths.WAD -
↪  _depositFeeRate(poolState_.rate));
}
```

But `debt < deposits` state aren't controlled for:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L311-L312

```
// check loan book's htp against new lup, revert if move drives LUP below HTP
if (params_.fromIndex < params_.toIndex && vars.htp > lup_) revert LUPBelowHTP();
```

As it's done in removeQuoteToken(), where it is `LUP < HTP || poolState_.debt > Deposits.treeSum(deposits_)`:

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L413-L425

```
lup_ = Deposits.getLup(deposits_, poolState_.debt);

uint256 htp = Maths.wmul(params_.thresholdPrice, poolState_.inflator);

if (
    // check loan book's htp doesn't exceed new lup
    htp > lup_
    ||
    // ensure that pool debt < deposits after removal
    // this can happen if lup and htp are less than min bucket price and htp > lup (since LUP is capped at min bucket price)
    (poolState_.debt != 0 && poolState_.debt > Deposits.treeSum(deposits_))
) revert LUPBelowHTP();
```

LUP is being bounded by deposits tree, i.e. the calculation assumes that total debt (the amount whose index is being located) is lower than total deposits (the tree where it is being located):

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/internal/Deposits.sol#L411-L422

```
/**
 *  @notice Returns `LUP` for a given debt value (capped at min bucket price).
 *  @param  deposits_ Deposits state struct.
 *  @param  debt_     The debt amount to calculate `LUP` for.
 *  @return `LUP` for given debt.
 */
function getLup(
    DepositsState storage deposits_,
    uint256 debt_
) internal view returns (uint256) {
    return _priceAt(findIndexOfSum(deposits_, debt_));
}
```

Tool used

Manual Review

Recommendation

Consider adding the `(poolState_.debt != 0 && poolState_.debt > Deposits.treeSum(deposits_))` logic to moveQuoteToken():

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L311-L312

```
        // check loan book's htp against new lup, revert if move drives LUP
↪  below HTP
-       if (params_.fromIndex < params_.toIndex && vars.htp > lup_) revert
↪  LUPBelowHTP();
+       if (params_.fromIndex < params_.toIndex && (vars.htp > lup_ ||
↪  (poolState_.debt != 0 && poolState_.debt > Deposits.treeSum(deposits_)))
↪  revert LUPBelowHTP();
```

The same approach can be added to HTP check in the kickWithDeposit() case.

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/901

dmitriia

> ajna-finance/contracts#901

Looks ok

Issue M-5: Wrong Inflator used in calculating HTP to determine accrualIndex in accrueInterest

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/111

Found by

Chinmay

Summary

When accruing Interest, the interest is added according to the deposits in all buckets upto the lower of LUP and HTP prices' buckets. But HTP is wrongly calculated in the `accrueInterest` function.

Vulnerability Detail

All major functions in `Pool.sol` use `_accrueInterest` which makes a call to `PoolCommons.accrueInterest` to accrue the interest for all deposits above the minimum of LUP and HTP prices, which means upto the lower of the `LupIndex` and `HTPIndex` because indexes move in the opposite direction of prices. Here in `accrueInterest` function, the `accrualIndex` is calculated as the higher of LUPIndex and HTPIndex to make sure interest is calculated for all deserving deposits above `min(HTP price, LUP price)` ie. `max(LUP index, HTP index)`.

But the `accrualIndex` has been implemented incorrectly. The HTP is calculated using the `newInflator` which incorporates the newly accrued interest into the HTP calculation, whereas the LUP is calculated with old values. The `accrualIndex` is set to `max(LUP index, HTP index)`. then.

Assume that the LUP price > HTP price. So, LUP index < HTP index. Hence, for the `interestEarningDeposit` all the deposits above the HTP index will be considered. But the value of HTP index is wrong now because it is calculated using new Inflator which means that the new Interest has been added in calculation of HTP already and thus the derived HTP index will be lower in value(which means upper in the bucket system). Assume that still after this LUP index < HTP index

Now since the old LUP index and new HTP index is used in the `max(LUP index, HTP index)` function, and LUP index is still < HTP index (ie. LUP price > HTP price) thus the deposits that were between the old HTP index and the new HTP index have been left out of the `interestEarningDeposit` calculation.

I talked to the developers about this discrepancy between new HTP and old LUP being used, and they said `"I can see the argument in favor of using the prior inflator here to be totally precise."`

Also, one of them said, `"It should be computed using the debt prior to the interest accrual itself, as it's determining the the minimum amount of deposit onto which that interest would be applied"`

This means that the htp index has been underestimated because it has been made lower(ie. upper in the bucket system) and thus the deposits that lie between the old HTP index and new HTP index have not been considered for calculating `interestEarningDeposit` when they should have been considered because before the interest accrual itself, those deposits were under the deserving `max(LUP index, HTP index)` bracket.

This means `interestEarningDeposit` has been underestimated and later calculations at PoolCommons.sol#L253 for lender Factor have become wrong.

Impact

This is a logic mistake and leads to wrong values for the lenderFactor.

Code Snippet

https://github.com/sherlock-audit/2023-04-ajna/blob/e2439305cc093204a0d927aac19d898f4a0edb3d/ajna-core/src/libraries/external/PoolCommons.sol#L232

Tool used

Manual Review

Recommendation

Calculate htp using the old inflator to correctly include all deposits that were deserving to get into `interestEarningDeposit` calculation.

Discussion

0xffff11

Deleted duplication of #88 due to explaining a slightly different issue

dmitriia

Fix in PR#916 looks ok, it replaces `newInflator_` with the current `poolState_.inflator` in accrueInterest()'s `htp` calculation.

Issue M-6: KickerActions uses wrong check to prevent Kickers from using deposits below LUP for KIckWithDeposit

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/113

Found by

Chinmay

Summary

The `kickWithDeposit` function in `KickerActions` has a check to prevent users having deposits below the LUP to use those deposits for kicking loans, but this check is implemented incorrectly.

Vulnerability Detail

The mentioned check evaluates if the `bucketPrice` used by the kicker is below the LUP. But the problem here is that the LUP used in this check is the new LUP that is calculated after incorporating the removal of the deposit itself and the debt changes. Thus, this check can be easily bypassed because the new LUP is bound to move lower and thus may cross past the `bucketPrice` used.

Consider a situation :

1. The kicker has deposits in a bucketPrice below LUP
2. Now he calls kickWithDeposit using this deposit
3. The new LUP is calculated after removing this deposit and adding the debt changes for kickPenalty etc.
4. This will make the LUP decrease and thus now LUP < bucketPrice that the user input

This way this check can be bypassed. According to the developers, the check was in place to prevent kickers from using deposits below the LUP to kick loans but this is not fulfilled.

Impact

This breaks protocol functionality because now anyone can deposit below LUP and use that to kick valid user loans. The `kickWithDeposit` function was only made to help depositors get their deposits back if some loans are blocking the withdrawl due to the LUP movement. But the current implementation allows anyone to kick those loans that were originally not eligible for liquidation.

This is a high severity issue because it griefs users off their funds by liquidating them, even when they were not eligible for liquidation.

Code Snippet

https://github.com/sherlock-audit/2023-04-ajna/blob/e2439305cc093204a0d927aac19d898f4a0edb3d/ajna-core/src/libraries/external/KickerActions.sol#L216

Tool used

Manual Review

Recommendation

Refactor the code and move this check to the top of the kickWithDeposit function logic.

Discussion

hrishibhat

Lead Watson comment:

> Invalid as removal deposit from the bucket below the LUP can't make LUP move, i.e. it can't be vars.bucketPrice < kickResult_.lup before removal from bucket, but vars.bucketPrice >= kickResult_.lup after that.

chinmay-farkya

Escalate for 10 USDC The lead watson's comment "removal deposit from the bucket below the LUP can't make LUP move" is incorrect.

Following up with the explanation above, see that at KickerActions#216 the LUP that is used for the check has been calculated at KickerActions#207. Here, the DepositState "deposits_" is the actual old state of deposits which means that kicker's deposit is still stored at whatever index it was. This deposit is actually only removed from "deposits_" at KickerActions#223 and KickerActions#241 which means the deposits used for kick is only removed from the "deposits_" after the check at Line 216.

Now look at the values in Line 207 : the LUP calculation using "deposits_" state where the user's deposit he is going to use to kick exists, but see the second function argument : `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)`

The LUP is being calculated as if the debt has increased by an amount, `amountToDebitFromDeposit` which is mirroring the removal of the deposit. But think about when the deposit index used (vars.bucketPrice) was originally below the

LUP. In this case, the use of increased debt in the calculation doesn't mirror the removal of deposits.

for example :

1. LUP = 100 and kicker uses bucket price 90 where he has 100K deposits.
2. Now this shouldn't move the LUP but it does move the LUP in calculation at Line 207 because it uses the same old Deposit but increased Debt(by whole of the deposits used for kick) in the calculation. Since LUP is calculated by summing up the deposits from top and matching it against the debt value passed to the getLup function, this will lower the LUP. This incorrect LUP will bypass the check and allow anyone to kick loans and grief users as explained in the original submission. This can be seen in Deposits.getLup function.

Because the logic can move the LUP below despite of a deposit from below the LUP being removed, this is a valid High severity finding as explained in impact above.

sherlock-admin

> Escalate for 10 USDC The lead watson's comment "removal deposit from the bucket below the LUP can’t make LUP move" is incorrect.
>
> Following up with the explanation above, see that at KickerActions#216 the LUP that is used for the check has been calculated at KickerActions#207. Here, the DepositState "deposits_" is the actual old state of deposits which means that kicker's deposit is still stored at whatever index it was. This deposit is actually only removed from "deposits_" at KickerActions#223 and KickerActions#241 which means the deposits used for kick is only removed from the "deposits_" after the check at Line 216.
>
> Now look at the values in Line 207 : the LUP calculation using "deposits_" state where the user's deposit he is going to use to kick exists, but see the second function argument : `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)`
>
> The LUP is being calculated as if the debt has increased by an amount, `amountToDebitFromDeposit` which is mirroring the removal of the deposit. But think about when the deposit index used (vars.bucketPrice) was originally below the LUP. In this case, the use of increased debt in the calculation doesn't mirror the removal of deposits.
>
> for example :
>
> 1. LUP = 100 and kicker uses bucket price 90 where he has 100K deposits.
> 2. Now this shouldn't move the LUP but it does move the LUP in calculation at Line 207 because it uses the same old Deposit but increased Debt(by whole of the deposits used for kick) in the

calculation. Since LUP is calculated by summing up the deposits from top and matching it against the debt value passed to the getLup function, this will lower the LUP. This incorrect LUP will bypass the check and allow anyone to kick loans and grief users as explained in the original submission. This can be seen in Deposits.getLup function.

Because the logic can move the LUP below despite of a deposit from below the LUP being removed, this is a valid High severity finding as explained in impact above.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

dmitriia

Escalate for 10 USDC Despite some misunderstandings, the take that `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)` is incorrect when `bucketPrice` is initially below LUP is a right one, i.e. the control do not filter out some `bucketPrice < initial_LUP` situations as LUP for the check isn't calculated correctly in this case. However, the filtering of `bucketPrice < LUP` is a more a convenience approach. There is no impact here that justify high severity, so valid medium looks the most suitable in this case.

To clarify:

1. "removal deposit from the bucket below the LUP can’t make LUP move" is correct all the time, this follows from the definition of LUP, which is the last utilized price, while `bucketPrice < LUP` deposits aren't utilized.
2. The `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)` check isn't correct exactly because of (1)

sherlock-admin

> Escalate for 10 USDC Despite some misunderstandings, the take that `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)` is incorrect when `bucketPrice` is initially below LUP is a right one, i.e. the control do not filter out some `bucketPrice < initial_LUP` situations as LUP for the check isn't calculated correctly in this case. However, the filtering of `bucketPrice < LUP` is a more a convenience approach. There is no impact here that justify high severity, so valid medium looks the most suitable in this case.
>
> To clarify:
>
> 1. "removal deposit from the bucket below the LUP can’t make LUP

move" is correct all the time, this follows from the definition of LUP, which is the last utilized price, while `bucketPrice < LUP` deposits aren't utilized.

2. The `kickResult_.lup = Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit)` check isn't correct exactly because of (1)

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

chinmay-farkya

I'd like to elaborate the impact to justify High severity :

1. Firstly, this allows an attacker to intentionally deposit below the LUP and force illegal liquidations on users. Since the loan with max TP is kicked first in order, this may cause a material loss of funds for that borrower depending upon his borrowed amounts.
2. The attacker can do this repeatedly to grief multiple users one by one. This may cause a material loss of funds for multiple users.
3. The attacker controls the deposited amount ie. `amountToDebitFromDeposit` above based on how much LP he holds in that bucket. This way he can move the LUP (for the intermediate calculation) lower to any price he desires. This is because larger the `amountToDebitFromDeposit` more the LUP will move. This means he can forcefully liquidate users even if they were far from underwater.

0xffff11

There are several points were the watson was mistaken. I really appreciate the comment from the senior exposing them. I do agree with a medium in the current landscape, but I would like a third opinion from @grandizzy

grandizzy

Agree with @dmitriia explanation and medium severity.

chinmay-farkya

Hey @grandizzy with regards to the Senior Watson's comments, I think I have justified the severity enough in my comment here

Do have a look. Won't take this further.

Also, Hey @0xffff11 would like to ask for more clarification on why the points I made in the last comment above aren't fulfilling to high severity.

grandizzy

fixed with https://github.com/ajna-finance/contracts/pull/894

dmitriia

> https://github.com/sherlock-audit/2023-04-ajna/blob/e2439305cc093204a0d927aac19d898f4a0edb3d/ajna-core/src/libraries/external/KickerActions.sol#L216

Looks ok, as a result of refactoring (for this and other issues of this contest combined) `kickWithDeposit()` is now `lenderKick()` that performs `if (vars.bucketPrice < Deposits.getLup(deposits_, poolState_.debt)) revert PriceBelowLUP()` check on the initial pool state.

hrishibhat

Result: Medium Unique In addition to the comments above after further review and discussion of this issue, Considering this issue a valid medium based on to the following comments from the Lead Watson:

> Depositing below LUP is definitely possible in general and is not enabled by this bug, what happens here is that LUP for kicking determined as Deposits.getLup(deposits_, poolState_.debt + vars.amountToDebitFromDeposit) is incorrect for deposits below LUP (and for them only, it is correct whenever bucket price >= LUP). This is actually an intermediary bug, I discussed it with the team early in the contest, but didn't reported as it led to several others so the team decided to rewrite the function altogether (it's now much lighter lenderKick()), and this requirement was initially introduced as a function separation matter, i.e. there is nothing wrong in using kicking with a bucket deposit when vars.bucketPrice < kickResult_.lup, it is just designed to handle an another case of a depositor pretending to remove the deposit and kicking as if they done that. I.e. that's basically a helper function that can be replicated via others (withdraw, then kick).
>
> Why it is medium: first of all kicking cannot be sniped and is always done in queue, one can kick only Loans.getMax(loans_).borrower, there is no optionality here, so if there are lots of bad borrowers the attacker will have to kick them first anyway, they can't just reach a good one, that's not possible, and, most importantly, kicking borrowers that aren't supposed to be kicked is substantially unprofitable, as kicker have to post a bond, which is partially forfeited if resulting auction price (i.e. market price as market actors will participate there as long as it's profitable) is higher than this borrower's price. The loss of the kicker is proportional to this gap, i.e. there is a guaranteed (as long as there are other market participants) loss for the kicker is they kick healthy borrowers, and the healthier is the borrower, the bigger the loss.
>
> So yes, an attacker can kick with this function, but it will be really costly griefing and nothing else. I.e. they will be required to post substantial

bond and will lose a part of it, while borrower will not lose as they will be liquidated at market price, i.e. their position will be just closed at the current levels. I.e. nobody besides attacker will have any substantial losses, attacker will be penalized for kicking at out of the market levels, their penalty will be redistributed.

This way this can be categozed as costly griefing, so the probability of such attack is low, while damage cannot be made significant (i.e. there can be some medium damage for borrower if market is specifically illiquid, but active market is basically a precondition for oracle-less protocols, and this situation cannot be controlled by the attacker).

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- chinmay-farkya: accepted
- dmitriia: accepted