SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Ajna
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: October 13 - October 27, 2023
Prepared on: January 9, 2024

Introduction

A noncustodial, peer-to-pool, permissionless lending, borrowing and trading system that requires no governance or external price feeds to function.

Scope

Repository: ajna-finance/ajna-core

Branch: main

Commit: 14e8655948efdb84af1a7eb96083cf9bec09d98b

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
4	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
0xkaden

santipu_
CL001

Issue H-1: Reserves can be stolen by settling artificially created bad debt from them

Source: https://github.com/sherlock-audit/2023-09-ajna-judging/issues/71

Found by

hyh

It is possible to inflate LUP and pull almost all the collateral from the loan, then settle this artificial bad debt off reserves, stealing all of them by repeating the attack.

Vulnerability Detail

LUP is determined just in time of the execution, but `removeQuoteToken()` will check for `LUP > HTP = Loans.getMax(loans).thresholdPrice`, so the attack need to circumvent this.

Attacker acts via 2 accounts under control, one for lending part, one for borrowing, having no previous positions in the given ERC20 pool. Let's say that pool state is as follows: `100` units of quote token utilized, and another `100` not utilized, for simplicity let's say all `100` are available, no auctions and no bad debt.

1. Get 100 loan at TP = new HTP just below LUP (let's name this loan the manipulated_loan, ML), let's say LUP is such that `150` quote token units worth of collateral was provided (at market price)
2. Add quote funds with `amount = {total utilized deposits} = 200` to a bucket significantly above the market (let's name it ultra_high_bucket, UHB)
3. Remove collateral from the ML up to allowed by elevated LUP = UHB price, say it's 10x market one. I.e. almost all, say remove `140`, leaving `10` quote token units worth of collateral (at market price)
4. Lender kick ML (removing funds from UHB will push LUP low and lender kick will be possible) to revive HTP reading
5. Remove all from UHB except quote funds frozen by the auction, i.e. remove `100`, leave `100`

1-5 go atomically in one tx.

No outside actors will be able to immediately benefit from the resulting situation as: a. UHB remove quote token is blocked due to the frozen debt b. take is unprofitable while auction price is above market c. bucket take is unprofitable while auction price is above UHB price

At this point attacker accounting in net quote tokens worth is:

1. `+100, -150`
2. `-200`
3. `+140`
4.
5. `+100`

Attacker has net `10` units of own capital still invested in the pool.

6. Attacker wait for auction price reaching UHB price, calls `takeBucket` with `depositTake == true`. The call can go with normal gas price as outside actors will not benefit from the such call and there will be no competition. `10` quote tokens worth of collateral will be placed to UHB and `98.482` units of quote token removed to cover the debt, `1.518` is the kicker's reward
7. Attacker calls `settlePoolDebt()` which settles the remaining `1.518` of debt from pool reserves as ML has this amount of debt and no collateral
8. Attacker removes all the funds from UHB with both initial and awarded LP shares, receiving `10` quote tokens worth of collateral and `1.518` quote token units of profit

6-8 go atomically in one tx.

At this point attacker accounting in quote tokens is:

6. (+kicker reward of ML stamped NP vs auction clearing UHB price in LP form)
7.
8. `+11.518`

Attacker has net 0 units of own capital still invested in the pool, and receives `1.518` quote token units of profit.

Profit part is a function of the pool's state, for the number above, assuming `auctionPrice == thresholdPrice = UHB price` and `poolRate_ = 0.05`: `1.518 = bondFactor * 100 = (npTpRatio - 1) / 10 * 100 = (1.04 + math.sqrt(0.05) / 2 - 1) / 10 * 100`, `bpf = bondFactor = takePenaltyFactor`.

Impact

It can be repeated to drain the whole reserves from the pool over time, i.e. reserves of any pool can be stolen this way.

Code Snippet

LUP is the only guardian for removing the collateral:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/BorrowerActions.sol#L287-L307

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

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

>>          uint256 encumberedCollateral = Maths.wdiv(vars.borrowerDebt,
 ↪  result_.newLup);
            if (
                borrower.t0Debt != 0 && encumberedCollateral == 0 || // case
 ↪  when small amount of debt at a high LUP results in encumbered collateral
 ↪  calculated as 0
                borrower.collateral < encumberedCollateral ||
                borrower.collateral - encumberedCollateral <
 ↪  collateralAmountToPull_
            ) revert InsufficientCollateral();

            // stamp borrower Np to Tp ratio when pull collateral action
            vars.stampNpTpRatio = true;

            borrower.collateral -= collateralAmountToPull_;

            result_.poolCollateral -= collateralAmountToPull_;
        }
```

The root cause is that bad debt is artificially created, with lender, borrower and kicker being controlled by the attacker, bad debt is then settled from the reserves:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L143-L146

```
// settle debt from reserves (assets - liabilities) if reserves positive, round
 ↪  reserves down however
if (assets > liabilities) {
    borrower.t0Debt -= Maths.min(borrower.t0Debt, Maths.floorWdiv(assets -
 ↪  liabilities, poolState_.inflator));
}
```

Kicking will revive the HTP as `_kick()` removing target borrower from loans:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/KickerActions.sol#L340-L341

```
// remove kicked loan from heap
Loans.remove(loans_, borrowerAddress_, loans_.indices[borrowerAddress_]);
```

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/base/Pool.sol#L225-L249

```
    function removeQuoteToken(
        ...
    ) external override nonReentrant returns (uint256 removedAmount_, uint256
↪   redeemedLP_) {
        ...

        uint256 newLup;
        (
            removedAmount_,
            redeemedLP_,
            newLup
        ) = LenderActions.removeQuoteToken(
            ...
            RemoveQuoteParams({
                maxAmount:      Maths.min(maxAmount_, _availableQuoteToken()),
                index:          index_,
>>              thresholdPrice: Loans.getMax(loans).thresholdPrice
            })
        );
```

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/LenderActions.sol#L434-L436

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

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

Tool used

Manual Review

Recommendation

Consider introducing a buffer representing the expected kicker reward in addition to LUP, so this part of the loan will remain in the pool.

Discussion

kadenzipfel

Escalate Should be medium.

According to the judging docs:

- How to identify a high issue "3. Significant loss of funds/large profit for the attacker at a minimal cost."
- How to identify a medium issue "3. A material loss of funds, no/minimal profit for the attacker at a considerable cost"

The profit margin from the provided example is ~0.4% (1.518 / (150 quote tokens worth of collateral at ML + 200 quote tokens at UHB) * 100%). This is consistent with the medium issue criteria ("minimal profit").

The minimum cost of the attack is greater than the total utilized deposits of the pool (**Note**: it's noted that the UHB is funded with the total utilized deposits of the pool, but the actual amount used in the example is 200, 2x the total utilized deposits and actually equal to the total utilized **and** unutilized deposits of the pool). Also note that it's not possible to use a flashloan in this circumstance since the attack requires multiple transactions across two EOAs. This is also consistent with the medium issue criteria ("considerable cost").

sherlock-admin2

> Escalate Should be medium.
>
> According to the judging docs:
>
> - How to identify a high issue "3. Significant loss of funds/large profit for the attacker at a minimal cost."
> - How to identify a medium issue "3. A material loss of funds, no/minimal profit for the attacker at a considerable cost"
>
> The profit margin from the provided example is ~0.4% (1.518 / (150 quote tokens worth of collateral at ML + 200 quote tokens at UHB) * 100%). This is consistent with the medium issue criteria ("minimal profit").
>
> The minimum cost of the attack is greater than the total utilized deposits of the pool (**Note**: it's noted that the UHB is funded with the total utilized deposits of the pool, but the actual amount used in the example is 200, 2x the total utilized deposits and actually equal to the total utilized **and**

> unutilized deposits of the pool). Also note that it's not possible to use a flashloan in this circumstance since the attack requires multiple transactions across two EOAs. This is also consistent with the medium issue criteria ("considerable cost").

You've created a valid escalation!

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.

neeksec

Agree with Escalation to downgrade severity to `Medium`.

dmitriia

The use of flash loans is possible as borrower and lender can have the **same** address of the attacking smart contract, there is no restrictions in this regard. `0.4%` is the immediate, atomic profit margin. The funds that are required to stay in the contract for auction are `10`, not `200`. If flash loans are free, for example obtained from Ajna itself, where the fee is zero, the total profit for the attacker in abstract quote token units is `1.518 - {2 days funding for 10} - total gas`.

This can be quite significant, say if `1 unit` here is `USDC 10k`, the profit is `15.18k - USDC_rate*2.0/365 * 100k - gas`. If we assume that USDC borrow rate is `10%` and total gas is `1 ETH = USD 2k`, it becomes `USDC 15.18 - 0.1*2.0/365*100 - 2 = 13.12k`. There is basically no market risk involved here, so it's arbitrage profit significant enough to compensate attacker for the efforts. This is for one pool and the attacking contract can be immediately reused for all the others.

As reserves of any pool can be drained this way fully, the reserves from all pools on all chains can be stolen. Since reserves burning is core monetization mechanics for protocl token this basically means that Ajna token will be worthless once this vector becomes public. Also, lack of reserves over time will move the growing share of pools to be insolvent as reserves also serve for bad debt coverage. Insolvency will render the pools unusable. The actual severity here is critical, but given the lack of this grade it is marked as high.

kadenzipfel

> 0.4% is the immediate, atomic profit margin.

No. As you explained in your example, steps 1-5 are atomic and by the end of step 5 the attacker is at a loss of 10 units of collateral. Profit doesn't occur until the attacker completes steps 6-8, and only if auction price gets low enough for attacker to execute these steps.

> There is basically no market risk involved here

This is false. The attacker is at a 10 unit loss (10% of the entire total utilized deposits) for, by your estimation, 2 days. They only ever profit if the auction price reaches the UHB. The Ajna team thus has 2 days to assess a highly suspicious auction and prevent further losses, in which they can simply `take` the quote tokens before the attacker, griefing the attacker into a significant loss and disincentivizing the attack entirely.

In summary, regardless of flashloans, as noted in escalation:

- The cost of the attack is still high (10% of the entire utilized deposits)
- The attacker profit is low (best case ~13% by your example: 100k risk, 13.12k profit) or **possibly negative** if intervention.
- The loss of funds incurred by the pool is also quite low at a maximum of ~0.76% (1.518 quote tokens lost / total pool size of 200 quote tokens * 100%)

kadenzipfel

This issue should also be invalid until a valid proof of concept is provided considering this attack is highly complex and far from obvious. See:
In case of non-obvious issues with complex vulnerabilities/attack paths, Watson m ust submit a valid POC for the issue to be considered valid and rewarded.

dmitriia

Market risk is risk that stems from movements of the market, it is absent in this scenario. Ajna team has little means to prevent the attack as the protocol is permissionless, while `take()` requires paying for collateral at prices substantially above the market, i.e. it is a material external investment that will be needed for all the pools. Also, if anyone takes before the attacker it will benefit them substantially (they are also a borrower, which will have their collateral priced even higher than they manipulated it themselves), resulting in even higher profitability, which will not disincentivize the attack.

- `10` units is not a cost of attack, it is a short-term investment, which funding costs are taken into account,
- `13%` for `2 days` is `(1.13) ^ (365 / 2) - 1 = 4_862_010_984 * 100% APY` which is significantly above market rates and is attractive for the wide range of attackers,
- This is loss per step. The reserves will be drained in full as nothing prevents attack to be immediately repeated. Also, it can be carried out simultaneously for many pools.

Again, this is critical severity vulnerability, having no low probability prerequisites and inflicting direct loss for the Ajna holders and pool's depositors (when reserves are lacking bad debt is written off from user's deposits starting with HPB).

dmitriia

> This issue should also be invalid until a valid proof of concept is provided considering this attack is highly complex and far from obvious

Protocol team has already run PoC for this, @ith-harvey, please confirm.

Czar102

From my understanding, this bug makes it possible for the attacker to immediately create bad debt. Even if the earnings/losses are a small fraction of the whole TVL of the pool, it is extremely severe for a lending protocol. Planning to keep it a high severity issue unless I'm misinterpreting or missing something.

Czar102

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- kadenzipfel: rejected

ith-harvey

The following code changes resolve this issue:

- https://github.com/ajna-finance/contracts/pull/962
- https://github.com/ajna-finance/contracts/pull/1008

dmitriia

Fix via PRs 962 and 1008 looks ok, conditional on the protocol running keeper bot that performs settlement calls.

The attack described can still be successful when there is a lack of activity in the pool, i.e. if we suppose that there is no party besides the attacker who will run the settlement and force attacker's deposit position to cover for the attack induced bad debt, then the attack still can be carried out with profit. When settlement does happen the attack provides no profit, but since there is also no principal fund loss for the attacker and cost of attack is substantially lower than potential profit, it might be the case that for low activity pools there still be some attempts to hold the position for `144 hours` and close bad debt off the reserves fully. In order to remove such a possibility opting for an automated solution that runs settlement whenever market participants didn't during a substantial period for any reason, i.e. protocol team controlled settlement of the last resort, is advised.

jacksanford1

Sherlock notes that this issue is fixed, but it's contingent on Ajna ensuring a keeper bot runs to perform settlement calls.

ith-harvey

> Sherlock notes that this issue is fixed, but it's contingent on Ajna ensuring a keeper bot runs to perform settlement calls.

We don't believe that the resolution to this issue should require or mention the maintenance of a keeper to ensure loans are settled for the following reasons:

- This proposed exploit doesn't work game-theoretically. For the attack to succeed, the attacker has to make a significant investment of capital, all of which will be lost entirely if anyone else calls settle ahead of them. The mere existence of a sequence of events that leads to the outcome isn't enough -- many things could fail if the actors in the pools don't react/respond reasonably to pool events.
- There are actors with a natural direct incentive to call settle -- namely any actor looking to interact with the pool post-settlement. If the pool is extremely inactive, so that there is no actor with such a natural incentive, it's unlikely that there would be substantial reserves to attempt to capture via this attack.
- Ajna token holders would always have a collective incentive to settle the loan proportional to the reserves that are being threatened, albeit a diffuse incentive.
- If the attack were to succeed, the harm occurs to Ajna token holders who no longer can buy the reserves. Lenders, borrowers and other end users are not affected.

Ajna is designed to be fully decentralized, and there will not be a centralized keeper running, and we'd like the report to reflect that.

Issue M-1: Wrong auctionPrice used in calculating BPF to determine bond reward (or penalty)

Source: https://github.com/sherlock-audit/2023-09-ajna-judging/issues/16

Found by

CL001 According to the Ajna Protocol Whitepaper(section 7.4.2 Deposit Take),in example:

"Instead of Eve purchasing collateral using take, she will call deposit take. Note auction goes through 6 twenty minute halvings, followed by 2 two hour halvings, and then finally 22.8902 minutes (0.1900179029 * 120min) of a two hour halving. After 6.3815 hours (*620 minutes +2*120 minutes + 22.8902 minutes), the auction price has fallen to 312, 998. 784 · 2 ^ (6+2+0.1900179) =1071.77 which is below the price of 1100 where Carol has 20000 deposit. Deposit take will purchase the collateral using the deposit at price 1100 and the neutral price is 1222.6515, so the BPF calculation is: BPF = 0.011644 * 1222.6515-1100 / 1222.6515-1050 = 0.008271889129".

As described in the whiterpaper, in the case of user who calls Deposit Take, the correct approach to calculating BPF is to use bucket price(1100 instead of 1071.77) when auctionPrice < bucketPrice .

Vulnerability Detail

1.bucketTake() function in TakerActions.sol calls the _takeBucket()._prepareTake() to calculate the BPF.
https://github.com/sherlock-audit/2023-09-ajna/blob/87abfb6a9150e5df3819de58cbd972a66b3b50e3/ajna-core/src/libraries/external/TakerActions.sol#L416
https://github.com/sherlock-audit/2023-09-ajna/blob/87abfb6a9150e5df3819de58cbd972a66b3b50e3/ajna-core/src/libraries/external/TakerActions.sol#L688

2.In _prepareTake() function,the BPF is calculated using vars.auctionPrice which is calculated by _auctionPrice() function.

```
function _prepareTake(
        Liquidation memory liquidation_,
        uint256 t0Debt_,
        uint256 collateral_,
        uint256 inflator_
    ) internal view returns (TakeLocalVars memory vars) {
 ........
        vars.auctionPrice = _auctionPrice(liquidation_.referencePrice, kickTime);
        vars.bondFactor   = liquidation_.bondFactor;
```

```
        vars.bpf          = _bpf(
            vars.borrowerDebt,
            collateral_,
            neutralPrice,
            liquidation_.bondFactor,
            vars.auctionPrice
        );
```

3.The _takeBucket() function made a judgment after _prepareTake()

```
 // cannot arb with a price lower than the auction price
if (vars_.auctionPrice > vars_.bucketPrice) revert AuctionPriceGtBucketPrice();
// if deposit take then price to use when calculating take is bucket price
if (params_.depositTake) vars_.auctionPrice = vars_.bucketPrice;
```

so the root cause of this issue is that in a scenario where a user calls Deposit Take(params_.depositTake ==true) ,BPF will calculated base on vars_.auctionPrice instead of bucketPrice.

And then,BPF is used to calculate takePenaltyFactor, borrowerPrice , netRewardedPrice and bondChange in the _calculateTakeFlowsAndBondChange() functionand direct impact on the _rewardBucketTake() function.
https://github.com/sherlock-audit/2023-09-ajna/blob/87abfb6a9150e5df3819de58cbd972a66b3b50e3/ajna-core/src/libraries/external/TakerActions.sol#L724

https://github.com/sherlock-audit/2023-09-ajna/blob/87abfb6a9150e5df3819de58cbd972a66b3b50e3/ajna-core/src/libraries/external/TakerActions.sol#L640

```
 vars_ = _calculateTakeFlowsAndBondChange(
            borrower_.collateral,
            params_.inflator,
            params_.collateralScale,
            vars_
        );
.............
_rewardBucketTake(
            auctions_,
            deposits_,
            buckets_,
            liquidation,
            params_.index,
            params_.depositTake,
            vars_
        );
```

Impact

Wrong auctionPrice used in calculating BFP which subsequently influences the _calculateTakeFlowsAndBondChange() and _rewardBucketTake() function will result in bias .

Following the example of the Whitepaper(section 7.4.2 Deposit Take) BPF = 0.011644 * (1222.6515-1100 / 1222.6515-1050) = 0.008271889129 The collateral purchased is min{20, 20000/(1-0.00827) * 1100, 21000/(1-0.01248702772)* 1100)} which is 18.3334 unit of ETH .Therefore, 18.3334 ETH are moved from the loan into the claimable collateral of bucket 1100, and the deposit is reduced to 0. Dave is awarded LPB in that bucket worth 18. 3334 · 1100 · 0. 008271889129 = 166. 8170374 . The debt repaid is 19914.99407 DAI

Based on the current implementations: BPF = 0.011644 * (1222.6515-1071.77 / 1222.6515-1050) = 0.010175. TPF = 5/4*0.011644 - 1/4 *0.010175 = 0.01201125. The collateral purchased is 18.368 unit of ETH. The debt repaid is 20000 * (1-0.01201125) / (1-0.010175) = 19962.8974DAI Dave is awarded LPB in that bucket worth 18. 368 · 1100 · 0. 010175 = 205.58 .

So,Dave earn more rewards38.703 DAI than he should have.

As the Whitepaper says: " the caller would typically have other motivations. She might have called it because she is Carol, who wanted to buy the ETH but not add additional DAI to the contract. She might be Bob, who is looking to get his debt repaid at the best price. She might be Alice, who is looking to avoid bad debt being pushed into the contract. She also might be Dave, who is looking to ensure a return on his liquidation bond."

Code Snippet

https://github.com/sherlock-audit/2023-09-ajna/blob/87abfb6a9150e5df3819de58cbd972a66b3b50e3/ajna-core/src/libraries/external/TakerActions.sol#L416

Tool used

Manual Review

Recommendation

In the case of Deposit Take calculate BPF using bucketPrice instead of auctionPrice .

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

n33k commented:

> "`if (params_.depositTake) vars_.auctionPrice = vars_.bucketPrice;` made "

Czar102

It seems this issue was accepted and it was opened during the escalation period. The sponsor closed the issue after the escalation period. Since there were no escalations, I believe this is valid.

I will reopen the issue.

ith-harvey

Resolved here:

- https://github.com/ajna-finance/contracts/pull/970

dmitriia

Fix via PRs 970 and 997 looks ok, `BPF` logic was updated and the material part of the associated surface was closed.

Issue M-2: HPB may be incorrectly bankrupt due to use of unscaled value in `_forgiveBadDebt`

Source: https://github.com/sherlock-audit/2023-09-ajna-judging/issues/25

Found by

0xkaden

An unscaled value is used in place of where a scaled value should be used in the bankruptcy check in `_forgiveBadDebt`. This may cause the bucket to be incorrectly marked as bankrupt, losing user funds.

Vulnerability Detail

At the end of `_forgiveBadDebt`, we do a usual bankruptcy check in which we check whether the remaining deposit and collateral will be little enough that the exchange rate will round to 0, in which case we mark the bucket as bankrupt, setting the bucket lps and effectively all user lps as 0.

The problem lies in the fact that we use `depositRemaining` as part of this check, which represents an unscaled value. As a result, when computing whether the exchange rate rounds to 0 our logic is off by a factor of the bucket's scale. The bucket may be incorrectly marked as bankrupt if the unscaled `depositRemaining` would result in an exchange rate of 0 when the scaled `depositRemaining` would not.

Impact

Loss of user funds.

Code Snippet

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L485

```
// If the remaining deposit and resulting bucket collateral is so small that the
↪  exchange rate
// rounds to 0, then bankrupt the bucket.  Note that lhs are WADs, so the
// quantity is naturally 1e18 times larger than the actual product
// @audit depositRemaining should be a scaled value
if (depositRemaining * Maths.WAD + hpbBucket.collateral * _priceAt(index) <=
↪  bucketLP) {
    // existing LP for the bucket shall become unclaimable
    hpbBucket.lps            = 0;
    hpbBucket.bankruptcyTime = block.timestamp;
```

```
    emit BucketBankruptcy(
        index,
        bucketLP
    );
}
```

Tool used

Manual Review

Recommendation

Simply scale `depositRemaining` before doing the bankruptcy check, e.g. something like:

```
depositRemaining = Maths.wmul(depositRemaining, scale);
if (depositRemaining * Maths.WAD + hpbBucket.collateral * _priceAt(index) <=
↪  bucketLP) {
    // existing LP for the bucket shall become unclaimable
    hpbBucket.lps            = 0;
    hpbBucket.bankruptcyTime = block.timestamp;

    emit BucketBankruptcy(
        index,
        bucketLP
    );
}
```

Discussion

ith-harvey

Resolve here: https://github.com/ajna-finance/contracts/pull/971

dmitriia

Fix looks ok, remaining deposit quote funds are now scaled in `_forgiveBadDebt()` bankruptcy logic.

Issue M-3: First pool borrower pays extra interest

Source: https://github.com/sherlock-audit/2023-09-ajna-judging/issues/26

Found by

0xkaden

There exists an exception in the interest logic in which the action of borrowing from a pool for the first time (or otherwise when there is 0 debt) does not trigger the inflator to update. As a result, the borrower's interest effectively started accruing at the last time the inflator was updated, before they even borrowed, causing them to pay more interest than intended.

Vulnerability Detail

For any function in which the current interest rate is important in a pool, we compute interest updates by accruing with `_accruePoolInterest` at the start of the function, then execute the main logic, then update the interest state accordingly with `_updateInterestState`. See below a simplified example for `ERC20Pool.drawDebt`:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/ERC20Pool.sol#L130

```
function drawDebt(
    address borrowerAddress_,
    uint256 amountToBorrow_,
    uint256 limitIndex_,
    uint256 collateralToPledge_
) external nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

   ...

    DrawDebtResult memory result = BorrowerActions.drawDebt(
        auctions,
        deposits,
        loans,
        poolState,
        _availableQuoteToken(),
        borrowerAddress_,
        amountToBorrow_,
        limitIndex_,
        collateralToPledge_
    );
```

```
    ...

    // update pool interest rate state
    _updateInterestState(poolState, result.newLup);

    ...
}
```

When accruing interest in `_accruePoolInterest`, we only update the state if `poolState_.t0Debt != 0`. Most notably, we don't set `poolState_.isNewInterestAccrued`. See below simplified logic from `_accruePoolInterest`:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/base/Pool.sol#L552

```
// check if t0Debt is not equal to 0, indicating that there is debt to be
↪  tracked for the pool
if (poolState_.t0Debt != 0) {
    ...

    // calculate elapsed time since inflator was last updated
    uint256 elapsed = block.timestamp - inflatorState.inflatorUpdate;

    // set isNewInterestAccrued field to true if elapsed time is not 0,
↪  indicating that new interest may have accrued
    poolState_.isNewInterestAccrued = elapsed != 0;

    ...
}
```

Of course before we actually update the state from the first borrow, the debt of the pool is 0, and recall that `_accruePoolInterest` runs before the main state changing logic of the function in `BorrowerActions.drawDebt`.

After executing the main state changing logic in `BorrowerActions.drawDebt`, where we update state, including incrementing the pool and borrower debt as expected, we run the logic in `_updateInterestState`. Here we update the inflator if either `poolState_.isNewInterestAccrued` or `poolState_.debt == 0`.

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/base/Pool.sol#L686

```
// update pool inflator
if (poolState_.isNewInterestAccrued) {
    inflatorState.inflator       = uint208(poolState_.inflator);
    inflatorState.inflatorUpdate = uint48(block.timestamp);
```

```
// if the debt in the current pool state is 0, also update the inflator and
↪  inflatorUpdate fields in inflatorState
// slither-disable-next-line incorrect-equality
} else if (poolState_.debt == 0) {
    inflatorState.inflator       = uint208(Maths.WAD);
    inflatorState.inflatorUpdate = uint48(block.timestamp);
}
```

The problem here is that since there was no debt at the start of the function, `poolState_.isNewInterestAccrued` is false and since there is debt now at the end of the function, `poolState_.debt == 0` is also false. As a result, the inflator is not updated. Updating the inflator here is paramount since it effectively marks a starting time at which interest accrues on the borrowers debt. Since we don't update the inflator, the borrowers debt effectively started accruing interest at the time of the last inflator update, which is an arbitrary duration.

We can prove this vulnerability by modifying `ERC20PoolBorrow.t.sol:testPoolBorrowAndRepay` to skip 100 days before initially drawing debt:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/tests/forge/unit/ERC20Pool/ERC20PoolBorrow.t.sol#L94

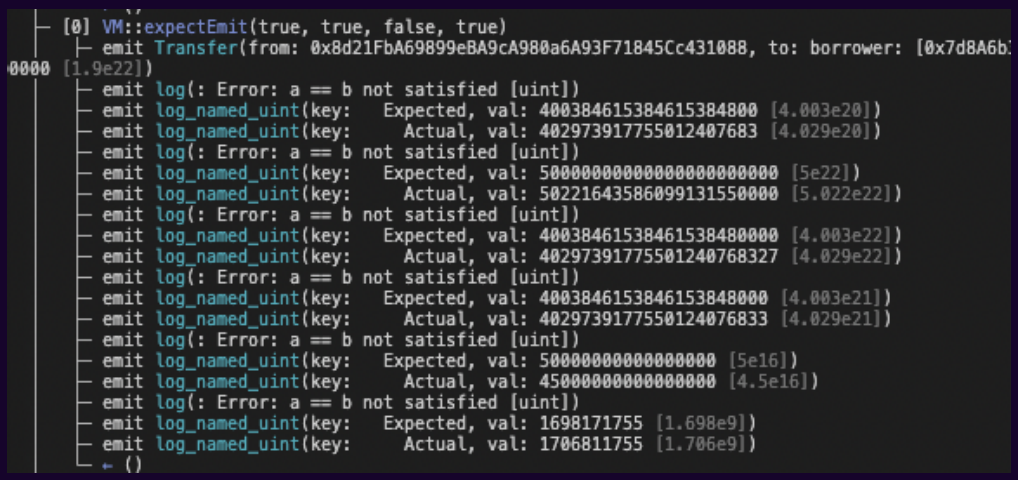
```
function testPoolBorrowAndRepay() external tearDown {
    // check balances before borrow
    assertEq(_quote.balanceOf(address(_pool)), 50_000 * 1e18);
    assertEq(_quote.balanceOf(_lender),        150_000 * 1e18);

    // @audit skip 100 days to break test
    skip(100 days);

    _drawDebt({
        from: _borrower,
        borrower: _borrower,
        amountToBorrow: 21_000 * 1e18,
        limitIndex: 3_000,
        collateralToPledge: 100 * 1e18,
        newLup: 2_981.007422784467321543 * 1e18
    });

    ...
}
```

Unlike the result without skipping time before drawing debt, the test fails with output logs being off by amounts roughly corresponding to the unexpected



interest.

Impact

First borrower **always** pays extra interest, with losses depending upon time between adding liquidity and drawing debt and amount of debt drawn.

Note also that there's an attack vector here in which the liquidity provider can intentionally create and fund the pool a long time before announcing it, causing the initial borrower to lose a significant amount to interest.

Code Snippet

See 'Vulnerability Detail' section for snippets.

Tool used

- Manual Review
- Forge

Recommendation

When checking whether the debt of the pool is 0 to determine whether to reset the inflator, it should not only check whether the debt is 0 at the end of execution, but also whether the debt was 0 before execution. To do so, we should cache the debt at the start of the function and modify the `_updateInterestState` logic to be something like:

```
// update pool inflator
if (poolState_.isNewInterestAccrued) {
    inflatorState.inflator       = uint208(poolState_.inflator);
    inflatorState.inflatorUpdate = uint48(block.timestamp);
// if the debt in the current pool state is 0, also update the inflator and
↪  inflatorUpdate fields in inflatorState
```

```
// slither-disable-next-line incorrect-equality
// @audit reset inflator if no debt before execution
} else if (poolState_.debt == 0 || debtBeforeExecution == 0) {
    inflatorState.inflator       = uint208(Maths.WAD);
    inflatorState.inflatorUpdate = uint48(block.timestamp);
}
```

Discussion

ith-harvey

Fixed here -> https://github.com/ajna-finance/contracts/pull/968

dmitriia

Fix looks ok, `inflatorUpdate` timestamp is now updated each time when `inflator == Maths.WAD`, while `inflator` is reset back to `Maths.WAD` when `poolState_.debt == 0`.

Issue M-4: Function `_indexOf` will cause a settlement to revert if `auctionPrice > MAX_PRICE`

Source: https://github.com/sherlock-audit/2023-09-ajna-judging/issues/30

Found by

santipu_

In ERC721 pools, when an auction is settled while `auctionPrice > MAX_PRICE` and borrower has some fraction of collateral (e.g. `0.5e18`) the settlement will always revert until enough time has passed so `auctionPrice` lowers below `MAX_PRICE`, thus causing a temporary DoS.

Vulnerability Detail

In ERC721 pools, when a settlement occurs and the borrower still have some fraction of collateral, that fraction is allocated in the bucket with a price closest to `auctionPrice` and the borrower is proportionally compensated with LPB in that bucket.

In order to calculate the index of the bucket closest in price to `auctionPrice`, the `_indexOf` function is called. The first line of that function is outlined below:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/helpers/PoolHelper.sol#L78

```
if (price_ < MIN_PRICE || price_ > MAX_PRICE) revert BucketPriceOutOfBounds();
```

The `_indexOf` function will revert if `price_` (provided as an argument) is below `MIN_PRICE` or above `MAX_PRICE`. This function is called from `_settleAuction`, here is a snippet of that:

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L234

```
function _settleAuction(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    address borrowerAddress_,
    uint256 borrowerCollateral_,
    uint256 poolType_
) internal returns (uint256 remainingCollateral_, uint256
↪  compensatedCollateral_) {
```

```
    // ...

            uint256 auctionPrice = _auctionPrice(
                auctions_.liquidations[borrowerAddress_].referencePrice,
                auctions_.liquidations[borrowerAddress_].kickTime
            );

            // determine the bucket index to compensate fractional collateral
>>>         bucketIndex = auctionPrice > MIN_PRICE ? _indexOf(auctionPrice) :
 ↪  MAX_FENWICK_INDEX;

    // ...
}
```

The `_settleAuction` function first calculates the `auctionPrice` and then it gets the index of the bucket with a price closest to `bucketPrice`. If `auctionPrice` results to be bigger than `MAX_PRICE`, then the `_indexOf` function will revert and the entire settlement will fail.

In certain types of pools where one asset has an extremely low market price and the other is valued really high, the resulting prices at an auction can be so high that is not rare to see an `auctionPrice > MAX_PRICE`.

The `auctionPrice` variable is computed from `referencePrice` and it goes lower through time until 72 hours have passed. Also, `referencePrice` can be much higher than `MAX_PRICE`, as outline in `_kick`:

```
vars.referencePrice = Maths.min(Maths.max(vars.htp, vars.neutralPrice),
 ↪  MAX_INFLATED_PRICE);
```

The value of `MAX_INFLATED_PRICE` is exactly `50 * MAX_PRICE` so a `referencePrice` bigger than `MAX_PRICE` is totally possible.

In auctions where `referencePrice` is bigger than `MAX_PRICE` and the auction is settled in a low time frame, `auctionPrice` will be also bigger than `MAX_PRICE` and that will cause the entire transaction to revert.

Impact

When the above conditions are met, the auction won't be able to settle until `auctionPrice` lowers below `MAX_PRICE`.

In ERC721 pools with a high difference in assets valuation, there is no low-probability prerequisites and the impact will be a violation of the system design, as well as the potential losses for the kicker of that auction, so setting severity to be high

Code Snippet

https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/SettlerActions.sol#L234

Tool used

Manual Review

Recommendation

It's recommended to change the affected line of `_settleAuction` in the following way:

```
-   bucketIndex = auctionPrice > MIN_PRICE ? _indexOf(auctionPrice) :
↪   MAX_FENWICK_INDEX;
+   if(auctionPrice < MIN_PRICE){
+       bucketIndex = MAX_FENWICK_INDEX;
+   } else if (auctionPrice > MAX_PRICE) {
+       bucketIndex = 1;
+   } else {
+       bucketIndex = _indexOf(auctionPrice);
+   }
```

Discussion

neeksec

Valid finding. Set to `Medium` because I don't see `the potential losses for the kicker of that auction`. This revert happens under extreme market condition and the it could recovery as time elapses.

dmitriia

Escalate

Since for `_settleAuction()` called outside of taking the exact time of settlement does not matter, the only remaining case is calling it in the taking workflow, and there the impact is that taker has to wait for the auction price to go under the limit. This is actually beneficial for the kicker since lower execution price means higher kicker's reward. Also, it is beneficial for the taker as they pay less, and the losing party is the borrower.

Given that `MAX_PRICE` is by definition set at a very high level that will not be achieved practically in the vast majority of the pools, the probability of the scenario is very low. Basically the only practical case I can see at the moment is that quote token defaults/breaks and its valuation goes to zero. In this case it will not matter

for the borrower at what exactly price the auction was settled. This way based on the very low probability and not straightforward materiality of the impact, that can be estimated as low/medium, I would categorize the overall severity to be low.

sherlock-admin2

> Escalate
>
> Since for `_settleAuction()` called outside of taking the exact time of settlement does not matter, the only remaining case is calling it in the taking workflow, and there the impact is that taker has to wait for the auction price to go under the limit. This is actually beneficial for the kicker since lower execution price means higher kicker's reward. Also, it is beneficial for the taker as they pay less, and the losing party is the borrower.
>
> Given that `MAX_PRICE` is by definition set at a very high level that will not be achieved practically in the vast majority of the pools, the probability of the scenario is very low. Basically the only practical case I can see at the moment is that quote token defaults/breaks and its valuation goes to zero. In this case it will not matter for the borrower at what exactly price the auction was settled. This way based on the very low probability and not straightforward materiality of the impact, that can be estimated as low/medium, I would categorize the overall severity to be low.

You've created a valid escalation!

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.

kadenzipfel

From the judging docs: "**Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue?** It would not count if the DOS, etc. lasts a known, finite amount of time <1 year."

santipu03

@dmitriia Agree that this issue will actually benefit the kicker and taker, making the borrower the losing party. Also agree that the impact is that takers will have to wait for `auctionPrice` to go lower than `MAX_PRICE`. And this can create the following scenario where a kicker unfairly profits from a borrower:

- Market price in a pool is higher than `MAX_PRICE` (only feasible on a few pools)
- Bob is a borrower that's collateralized enough so kicking him would cause a loss on kicker.

- A malicious lender can kick Bob knowing that the auction won't settle until `auctionPrice` is below `MAX_PRICE`, then the kicker will be receiving more profits from the bond, thus profiting off the borrower.
- Therefore, Bob will be receiving less tokens from the auction as expected, and the kicker will be pocketing a profit taking advantage of this bug.

On the other hand, the only pools affected by this issue would be pools with a high difference in valuation (e.g. `BTC/SHIBA`, `ETH/SHIBA`, etc). Because the difference of valuation of a unit of token on these pools is huge, it'd be pretty easy to achieve `MAX_PRICE`. For example:

- Market Price BTC/SHIBA: `4_295_487_804 * 1e18`
- Market Price ETH/SHIBA: `229_958_536 * 1e18`
- MAX_PRICE in Ajna = `1_004_968_987 * 1e18`

Achieving `MAX_PRICE` is not unfeasible in these sort of pools.

@kadenzipfel A DoS by itself doesn't qualify for a medium/high but in the case of this issue it also implies the loss of borrower funds.

Given that Ajna will be an immutable protocol that aims to support this kind of pools, the impact of this issue is high. And because these sort of pools only represents a minority, the probability of this happening is medium/low. Therefore, I think the fair severity for this issue is medium.

kadenzipfel

> A malicious lender can kick Bob knowing that the auction won't settle until auctionPrice is below MAX_PRICE, then the kicker will be receiving more profits from the bond, thus profiting off the borrower.

It seems what you're trying to argue here is that you can't `take` the full amount auctioned because it will attempt to settle the auction if the remaining `borrower_.t0Debt == 0`, see `TakerActions:L524-539` below:

```
// if debt is fully repaid, settle the auction
if (borrower_.t0Debt == 0) {
    settledAuction_ = true;

    // settle auction and update borrower's collateral with value after
↪   settlement
    (remainingCollateral_, compensatedCollateral_) =
↪   SettlerActions._settleAuction(
        auctions_,
        buckets_,
        deposits_,
        borrowerAddress_,
```

```
        borrower_.collateral,
        poolState_.poolType
    );

    borrower_.collateral = remainingCollateral_;
}
```

The flaw with your argument is that you can still `take` at whatever the current auction price is as long as you leave some `borrower_.t0Debt` so that we don't get the unexpected revert in `_settleAuction`. So the borrower could easily `take` most of the auctioned debt at a favourable price, resulting in a loss for the kicker, maintaining intended system incentives.

santipu03

> The flaw with your argument is that you can still take at whatever the current auction price is as long as you leave some borrower_.t0Debt so that we don't get the unexpected revert in _settleAuction. So the borrower could easily take most of the auctioned debt at a favourable price, resulting in a loss for the kicker, maintaining intended system incentives.

This isn't true in ERC721 pools because is required to take a minimum of one unit of collateral (`1e18`) for every `take` action.

Before calculating the amounts in `take`, collateral to take is rounded to only get a minimum of 1 unit: https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L358-L361

```
// for NFT take make sure the take flow and bond change calculation happens for
↪  the rounded collateral that can be taken
if (params_.poolType == uint8(PoolType.ERC721)) {
    takeableCollateral = (takeableCollateral / 1e18) * 1e18;
}
```

Also, after calculating the collateral to take, `collateralAmount` is rounded up in order to get full units of collateral (min: `1e18`): https://github.com/sherlock-audit/2023-09-ajna/blob/main/ajna-core/src/libraries/external/TakerActions.sol#L383-L402

```
if (params_.poolType == uint8(PoolType.ERC721)) {
    // slither-disable-next-line divide-before-multiply
    uint256 collateralTaken = (vars_.collateralAmount / 1e18) * 1e18; //
↪  solidity rounds down, so if 2.5 it will be 2.5 / 1 = 2

    // collateral taken not a round number
    if (collateralTaken != vars_.collateralAmount) {
```

```
        if (Maths.min(borrower_.collateral, params_.takeCollateral) >=
↪   collateralTaken + 1e18) {
            // round up collateral to take
            collateralTaken += 1e18;

            // taker should send additional quote tokens to cover difference
↪   between collateral needed to be taken and rounded collateral, at auction
↪   price
            // borrower will get quote tokens for the difference between rounded
↪   collateral and collateral taken to cover debt
            vars_.excessQuoteToken = Maths.wmul(collateralTaken -
↪   vars_.collateralAmount, vars_.auctionPrice);
            vars_.collateralAmount = collateralTaken;
        } else {
            // shouldn't get here, but just in case revert
            revert CollateralRoundingNeededButNotPossible();
        }
    }
}
```

Therefore, in ERC721 pools, the minimum amount of collateral to take is `1e18`, and this will prevent takers to take most of the collateral at a favourable price when collateral auctioned is low (e.g. 1 NFT).

dmitriia

> @dmitriia Agree that this issue will actually benefit the kicker and taker, making the borrower the losing party. Also agree that the impact is that takers will have to wait for `auctionPrice` to go lower than `MAX_PRICE`. And this can create the following scenario where a kicker unfairly profits from a borrower: ... On the other hand, the only pools affected by this issue would be pools with a high difference in valuation (e.g. `BTC/SHIBA`, `ETH/SHIBA`, etc). Because the difference of valuation of a unit of token on these pools is huge, it'd be pretty easy to achieve `MAX_PRICE`. For example:
>
> - Market Price BTC/SHIBA: `4_295_487_804 * 1e18`
> - Market Price ETH/SHIBA: `229_958_536 * 1e18`
> - MAX_PRICE in Ajna = `1_004_968_987 * 1e18`
>
> Achieving `MAX_PRICE` is not unfeasible in these sort of pools.

This means that until `MAX_PRICE` is increased `BTC/SHIBA` and alike pools will malfunction in Ajna, which needs to be communicated as pool creation is permissionless.

Re ERC721 it looks like the attack is for `1e18` amount only as before that takes can

avoid `_settleAuction()` by leaving this `1` NFT intact.

santipu03

> Re ERC721 it looks like the attack is for 1e18 amount only as before that takes can avoid _settleAuction() by leaving this 1 NFT intact.

The attack is most effective for auctions of `1e18` collateral, but it's still efective (though less impactful) with auctions for more amount of collateral.

For example, in case of an auction of `2.1e18` collateral the borrower will receive the fair price for 1 NFT but a lower price for the other NFT. This unfair price for the last NFT of an auction can be the difference between a kicker winning over a bond or losing part or it.

As said, the probability is kind of low but the impact is high so I think the fair severity is medium.

Evert0x

@kadenzipfel @dmitriia do you have any follow up arguments? If not I will keep the issue state as is.

dmitriia

> @kadenzipfel @dmitriia do you have any follow up arguments? If not I will keep the issue state as is.

I'm fine with medium as issue highlights the important enough limitations (first of all, `MAX_PRICE` looks to be too low) for extreme pairs of tokens, which are relevant due to permissionless nature of Ajna pools.

Czar102

To me it looks like a limitation introduced by `MAX_PRICE`, which seems to be a design choice, hence should be out of scope. @santipu03 please let me know if that's accurate. If yes, I will be accepting the escalation and closing the issue.

santipu03

I don't think this issue is because of a design choice. In ERC20 pools, taking collateral from auctions with `auctionPrice > MAX_PRICE` is always possible. This is also true in ERC721 pools when dealing with integer collateral (e.g., `1e18`). However, the problem arises specifically in ERC721 pools with fractioned collateral (e.g., `1.5e18`). This inconsistency points to the issue being a bug, not an intended feature.

If it were a design choice, the issue would appear in all pool types under any scenario. Instead, it only happens in ERC721 pools with fractioned collateral, further indicating it's an unintentional bug.

The protocol allows for takers to take collateral in an auction with a price above `MAX_PRICE`, therefore this issue is not a design choice but a bug. The fix proposed in

the report addresses and corrects this issue, reinforcing the idea that it's a bug and not a designed aspect of the protocol.

Czar102

Sorry for my misunderstanding. Thank you for the explanation. I plan on rejecting the escalation and leaving the issue as is.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- dmitriia: rejected

ith-harvey

After checking using this test -> https://github.com/ajna-finance/contracts/blob/f7b9f83e0f1048390cb986b0530999e12fc784a4/tests/forge/unit/ERC721Pool/ERC721PoolLiquidationsHighTake.t.sol#L160 I do not consider this an issue.

In an ERC721 pool when take() is called and the auction price exceeds the MAX_PRICE (highest price bucket price) and the collateral amounts tie out the price check logic inside of _settleAuction() is not run due to this check here: https://github.com/ajna-finance/ajna-core/blob/14e8655948efdb84af1a7eb96083cf9bec09d98b/src/libraries/external/SettlerActions.sol#L223

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.