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 15, 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
7	7

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

Bauchibred	GimelSec	Chinmay
ctf_sec	branch_indigo	
stopthecap	hyh	

Issue H-1: `permit` signatures can be replayed if approval is revoked during valid timestamp

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/46

Found by

ctf_sec, stopthecap

Summary

permit signatures can be replayed if approval is revoked during valid timestamp

Vulnerability Detail

When using the `permit` function in the `PermitERC721` contract, there is a flaw when a signature can be replayed to steal NFTs from the owner.

The attack vector opens when an owner (Bob) let's say of NFT id `1` signs a permit for address `0x01` with whatever deadline that is not `block.timestamp`. If Bob revokes the approval for `0x01`, meanwhile the timestamp still holds:

the signature will be able to be replayed by anyone, most likely `0x01` to approve the spending of the token again.

This happens because the `nonce` that they use to generate the `digest` :

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/PermitERC721.sol#L147

is only updated if the token is actually transferred, enabling to replay the signature if the approval is revoked.

Impact

Steal an NFT from a previous canceled approval

Code Snippet

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/PermitERC721.sol#L133-L157

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/PermitERC721.sol#L269

https://github.com/sherlock-audit/2023-04-ajna/blob/main/ajna-core/src/base/PermitERC721.sol#L147

Tool used

Manual Review

Recommendation

Increment the `nonce` on approvals too, not just on transfers. It should be a new nonce for every call to permit.

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/906

dmitriia

> ajna-finance/contracts#906

Looks ok

mohammedrizwann123

Escalate for 10 USDC.

This finding and #35 are both invalid.

In ERC721Permit, nonce is incremented in case of transfer only. nonce is not incremented in permit().

Reason:-

> Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:

Reference link- https://soliditydeveloper.com/erc721-permit

EIP4494 link- https://eips.ethereum.org/EIPS/eip-4494

Reference implementation link- https://github.com/dievardump/erc721-with-permits/blob/main/contracts/ERC721WithPermit.sol

This is an already discussed issue with @grandizzy and it is confirmed that this Reference implementation contract is used which is absolutely correct.

sherlock-admin

> Escalate for 10 USDC.
>
> This finding and #35 are both invalid.
>
> In ERC721Permit, nonce is incremented in case of transfer only. nonce is not incremented in permit().
>
> **Reason:-**
>
> > Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:
>
> Reference link- https://soliditydeveloper.com/erc721-permit
>
> EIP4494 link- https://eips.ethereum.org/EIPS/eip-4494
>
> Reference implementation link- https://github.com/dievardump/erc721-with-permits/blob/main/contracts/ERC721WithPermit.sol
>
> This is an already discussed issue with @grandizzy and it is confirmed that this Reference implementation contract is used which is absolutely correct.

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.

ctf-sec

Escalate for 10 USDC.

Referenced implementation does not means correct implementation.

> Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:

this is a feature, but the report describe a vulnerability with impact that can rug user (the user protection does have high priority as we seen in other finding) (signature replay)

even not incrementing nonce in permit, the same signature should not be used twice

so this report is still a valid high

sherlock-admin

> Escalate for 10 USDC.
>
> Referenced implementation does not means correct implementation.
>
> > Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:
>
> this is a feature, but the report describe a vulnerability with impact that can rug user (the user protection does have high priority as we seen in other finding) (signature replay)
>
> even not incrementing nonce in permit, the same signature should not be used twice
>
> so this report is still a valid high

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.

mohammedrizwann123

@ctf-sec Thank you for the comment.

I don't agree with the escalation and still find both issues as invalid. Please refer below additional context in continuation to my above escalation.

ERC-2612(ERC20Permit) and ERC-4494(ERC721Permit) are completely different as far as nonce implementation is concerned for signature reply attacks.

I understand with @ctf-sec "the signature will be able to be replayed by anyone, most likely 0x01 to approve the spending of the token again." that **this is possible**

with ERC-2612(ERC20Permit) and signature replay attack is possible. In this case nonce mapping is given as below to mitigate the signature replay attacks with increment in ERC-2612(ERC20Permit),

```
mapping(address => uint256) private _nonces;
```

Here the nonce is linked with address. Thats where the "signature replay attack prevention is needed".

However, if you see the ERC-4494(ERC721Permit) the nonce mapping is implemented as below,

```
24    mapping(uint256 => uint256) private _nonces;
```

Here the nonce is linked with tokenId. This should not be confused with nonce(address). It can be said nonce in ERC-4494 is not dealing with addresses(owner or user) but it is dealing with tokenId. Therefore the signature attack assumption is not valid for nonce in this case. This can be seen in the actual implementation at L-105 as below,

```
93    function permit(
94        address spender,
95        uint256 tokenId,
96        uint256 deadline,
97        bytes memory signature
98    ) public {
99        require(deadline >= block.timestamp, '!PERMIT_DEADLINE_EXPIRED!');
100
101        bytes32 digest = _buildDigest(
102            // owner,
103            spender,
104            tokenId,
105            _nonces[tokenId],
106            deadline
107        );
```

Referring to ERC-4494 discussion at ethereum-magicians,

> ERC2612 takes the value being approved as an argument, and increments a nonce after each call to the permit function in order to prevent replay attacks. Since each ERC721 token is discrete, it allows for having the nonce based not on the owner's address or calls to permit, but rather to tie the nonce to tokenId and to increment on each transfer of the NFT.

nonces(uint256) is not the same as nonces(address).

EIP4494 - Authors | Simon Fremaux (dievardump), William Schwab (wschwab) have discussed lots of issues in below link. https://ethereum-magicians.org/t/eip-4494-extending-erc-2612-style-permits-to-erc-721-nfts/7519/4

I highly recommend to have detail look at this discussion on EIP4494.

In addition, Please go through the eip-4494 to know why this separate EIP is proposed, how its design architecture is different with ERC20Permit and why signature attack is not possible with EIP-4494
https://eips.ethereum.org/EIPS/eip-4494

I believe the implementation used by @grandizzy is correct with no issues.

With due respect to judge, Final decision by judge should be agreed.

Thank you.

ww4tson

Revoking the approval in OZ ERC721 can be done in 3 ways:

- approve

```
function _approve(address to, uint256 tokenId) internal virtual {
    _tokenApprovals[tokenId] = to;
    emit Approval(ownerOf(tokenId), to, tokenId);
}
```

- transferFrom/safeTransferFrom

```
function _transfer(address from, address to, uint256 tokenId) internal virtual {
    address owner = ownerOf(tokenId);
    if (owner != from) {
        revert ERC721IncorrectOwner(from, tokenId, owner);
    }
    if (to == address(0)) {
        revert ERC721InvalidReceiver(address(0));
    }

    _beforeTokenTransfer(from, to, tokenId, 1);

    // Check that tokenId was not transferred by `_beforeTokenTransfer` hook
    owner = ownerOf(tokenId);
    if (owner != from) {
        revert ERC721IncorrectOwner(from, tokenId, owner);
    }

    // Clear approvals from the previous owner
    delete _tokenApprovals[tokenId];
```

- burn

```
function _burn(uint256 tokenId) internal virtual {
    address owner = ownerOf(tokenId);

    _beforeTokenTransfer(owner, address(0), tokenId, 1);

    // Update ownership in case tokenId was transferred by
↪   `_beforeTokenTransfer` hook
    owner = ownerOf(tokenId);

    // Clear approvals
    delete _tokenApprovals[tokenId];
```

ctf-sec

I agree wtih ww4tson that the revoke can be done in three ways and the first way is the only way to revoke or change token id approval without changing ownership

```
function _approve(address to, uint256 tokenId) internal virtual {
      _tokenApprovals[tokenId] = to;
      emit Approval(ownerOf(tokenId), to, tokenId);
  }
```

the auditor mohammedrizwann123 points out the nonce maps to the token id

```
mapping(address => uint256) private _nonces;
```

and

> Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:

this describes the feature, but the bug report describes a vulnerability that can harm end user with prove.

In ERC20 permit, if signature is replayed, ERC20 token can be stolen

if ERC721 permit, if the signature is replayed, the ERC721 token can be stoken

In fact, mohammedrizwann123 quote this link:

https://ethereum-magicians.org/t/eip-4494-extending-erc-2612-style-permits-to-erc-721-nfts

there is a comment in the post

> If the use case is thinking that two signed messages will be executed together, e.g. 1 permit message and 1 order, well there is an assumption they "have" to be processed together, which is a false assumption. Unless they are committed to the blockchain (e.g. The permit is "used up") it is still valid, and can be replayed in any venue that will accept it. You could let it linger for longer, and have permits for multiple such contracts at the same time, but the longer it's valid for, the more likely it is that you'll forget about signing the message, which can come around and bite you at a later point in time a la BAYC "thefts".

so this report is valid

------ if the judge and sherlock agree, no need to read further, otherwise, I wrote a POC to show how after the owner explicitly change the token spender, the permit signature can still be replayed to steal NFT from the owner ------

the fully running POC is here:

https://drive.google.com/file/d/11JW6w78l1_Ytzb-tHM-wsnl6g5IleTgE/view?usp=sharing

note, I add a mint method to mint NFT and make the _buildDigest modifier level public for POC

the POC is

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.14;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import "../src/ERC721Permits.sol";


contract CounterTest is Test {

    PermitERC721 nft = new PermitERC721("name", "symbol", "1");
    uint256 creatorPrivateKey = 5201314;

    function setUp() public {


    }
    function getEthSignedMessageHash(
        bytes32 _messageHash
    ) public pure returns (bytes32) {
        /*
        Signature is produced by signing a keccak256 hash with the following
↪   format:
        "\x19Ethereum Signed Message\n" + len(msg) + msg
        */
        return
            keccak256(
                abi.encodePacked("\x19Ethereum Signed Message:\n32",
↪   _messageHash)
            );
    }

    function testReplayApproval() public {

        // mint NFT
        address ownerAddress = 0x0BACD98d94DCDDC049B5d52479aDf6896bb63686;
        nft.mint(ownerAddress, 1);

        // prepare field for signature
        // spender, tokenId, nonce and deadline
        uint256 deadline = block.timestamp + 3600;

        uint256 tokenId = 1;

        address spender1 = vm.addr(111);

        address spender2 = vm.addr(222);

        uint256 nonce = nft.nonces(1);

        bytes32 proveHash = nft._buildDigest(spender1, tokenId, nonce, deadline);

        bytes32 digest = getEthSignedMessageHash(proveHash);

        // sign the signature
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(creatorPrivateKey, digest);
        bytes memory signature = abi.encodePacked(r, s, v);

        // give spender1's permit approval
        nft.permit(spender1, tokenId, deadline, signature);

        address operator = nft.getApproved(tokenId);

        // now we verify the spender1 is the spender of the token 1 for nft owner
        console.log("operator is the spender1,", operator == spender1);

        // nft owner decides to revoke the approval of the spender1
        // and now the spender2 has the spending power of the token1
        vm.prank(ownerAddress);
        nft.approve(spender2, tokenId);

        operator = nft.getApproved(tokenId);

        console.log("operator is the spender2,", operator == spender2);

        // signature replay
        console.log("replay the signature, spender1 becomes the spender again");
        nft.permit(spender1, tokenId, deadline, signature);

        operator = nft.getApproved(tokenId);
        console.log("operator is the spender1, not spender2,", operator ==
 ↪  spender1);

        vm.prank(spender1);
        nft.transferFrom(ownerAddress, spender1, tokenId);

        console.log("spender1 steal nft", nft.ownerOf(tokenId) == spender1);


    }

}
```

we run

```
forge test -vvv
```

and the output

```
Running 1 test for test/Counter.t.sol:CounterTest
[PASS] testReplayApproval() (gas: 138663)
Logs:
  operator is the spender1, true
  operator is the spender2, true
  replay the signature, spender1 becomes the spender again
  operator is the spender1, not spender2, true
  spender1 steal nft true
```

prateek105

> If the use case is thinking that two signed messages will be executed together, e.g. 1 permit message and 1 order, well there is an assumption they "have" to be processed together, which is a false assumption. Unless they are committed to the blockchain (e.g. The permit is "used up") it is still valid, and can be replayed in any venue that will accept it. You could let it linger for longer, and have permits for multiple such contracts at the same time, but the longer it's valid for, the more likely it is that you'll forget about signing the message, which can come around and bite you at a later point in time a la BAYC "thefts".

The same person who has mentioned above issue here https://ethereum-magicians.org/t/eip-4494-extending-erc-2612-style-permits-to-erc-721-nfts/7519/28 has stated right after this paragraph that `Either way, I don't think this is a big issue.`

The signature can be replayed but this is a required behaviour and the token owner should think before providing permit signatures and also use the `deadline` parameter wisely. In my opinion as mentioned in the discussion thread for eip 4494, multiple permit feature is much more important for eg: There can be a case when the user permits a contract (say LiquidityManager) to perform moveLiquidity on his behalf and also lists NFT on a marketplace to sell in an auction. So if at any time LiquidityManager calls permit to get approval to call moveLiquidity, the signature for the marketplace becomes invalid, and the user will need to make a new signature again which should not be the case as he might not be able to sell his NFT.

JeffCX

> > If the use case is thinking that two signed messages will be executed together, e.g. 1 permit message and 1 order, well there is an assumption they "have" to be processed together, which is a false assumption. Unless they are committed to the blockchain (e.g. The permit is "used up") it is still valid, and can be replayed in any venue that will accept it. You could let it linger for longer, and have permits for multiple such contracts at the same time, but the longer it's valid for, the more likely it is that you'll forget about signing the message, which can come around and bite you at a later point in time a la BAYC "thefts".
>
> The same person who has mentioned above issue here https://ethereum-magicians.org/t/eip-4494-extending-erc-2612-style-permits-to-erc-721-nfts/7519/28 has stated right after this paragraph that `Either way, I don't think this is a big issue.`
>
> The signature can be replayed but this is a required behaviour and the token owner should think before providing permit signatures and also use the `deadline` parameter wisely. In my opinion as mentioned in the discussion thread for eip 4494, multiple permit feature is much more important for eg: There can be a case when the user permits a contract

> (say LiquidityManager) to perform moveLiquidity on his behalf and also lists NFT on a marketplace to sell in an auction. So if at any time LiquidityManager calls permit to get approval to call moveLiquidity, the signature for the marketplace becomes invalid, and the user will need to make a new signature again which should not be the case as he might not be able to sell his NFT.

the impact described the comment above is saying the protocol should support additional feature.

> the signature for the marketplace becomes invalid, and the user will need to make a new signature again which should not be the case as he might not be able to sell his NFT.

while the report with code POC proof shows how to replay the signature to potentially steal NFT.

the escalation keep describing a feature and does not explain why the report is invalid

0xffff11

I do think this is a valid issue. I have read about the mentioned eip and everything, but either might not be the best decision for this protocol or it is incorrectly used, but as shown by @JeffCX on his/her PoC the attack is possible and the impact described is correct.

dmitriia

Let me clarify:

The change for this was reverted with the following reasoning:

Explanation of the EIP-4494 design (Mohammed's post):

> Reason why nonce is not required in permit(),
>
> > Well because with permits and NFT's you actually have a unique feature opportunity that's not possible with ERC20. You can allow a user to create multiple permit signatures for multiple spender addresses all for the same tokenId. All spenders can execute the permit function using the same nonce since we aren't incrementing the nonce inside permit. And only if the NFT is actually transferred, the nonce is incremented making the old signatures invalid. So make sure in your ERC721 contract to increase the nonce for every transfer:

Discussion (my posts):

> I.e. one can list a NFT on several exchanges and one which has the first bid will perform the actual transfer. This is a feature conditional on the

> fact that all exchange contracts are trusted, so the owner knows that the approval will be utilized only having the payment transfer (bid) tied to it.
>
> Exchanges aren't relevant for position NFTs, but various asset management solutions might be. For example, one might to give permits to one contract that keep the deposit 10% away from the market, and another to a contract that reinvest the assets somewhere else when opportunity arises (i.e. one moves the position, another redeems it, each having own logic, some independent solutions).

dmitriia

To rephrase, while it is true that permits issued to semi-trusted actors (say EOAs, whose priorities can change) are exploitable in this design, it is a trade-off for the ability to give various services (contracts with well-defined roles) the straightforward way to perform a one time action, say fulfil a bid in the case of exchanges.

One do trust an exchange in this setup and the need for such trust is a known limitation of the approach.

ctf-sec

Then sounds like a high severity with sponsor confirmed and won't fix is reasonable. report itself + POC is valid :) all comments doesn't really point out and say the report is invalid....

dmitriia

There is a burden of proof belonging to the reporter, not the other way around. Here the use case demonstrated (permit for EOA turning malicious) is not actually relevant, position NFTs basically would not be exchanged between EOAs.

ctf-sec

I agree the reportor carries the burden of proof

```

vm.prank(spender1);
nft.transferFrom(ownerAddress, spender1, tokenId);

console.log("spender1 steal nft", nft.ownerOf(tokenId) == spender1);
```

the spender1 could be a contract or a EOA, does not have to be an EOA, and anyone (including an EOA) can replay the signature to given approval to the contract

MLON33

> > ajna-finance/contracts#906
>
> Looks ok

Moving sign-off by @dmitriia here for clarity.

Issue H-2: 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-3: 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();
```

SHERLOCK

```
    // 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-4: 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.lupt0DebtEma = uint256(
    PRBMathSD59x18.mul(vars.weightTu, int256(vars.lupt0DebtEma)) +
    PRBMathSD59x18.mul(1e18 - vars.weightTu, int256(interestParams_.lupt0Debt))
);
```

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 lupt0DebtEma_
) 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 = (lupt0DebtEma_ != 0) ?
        int256(Maths.wdiv(debtColEma_, lupt0DebtEma_)) : 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-5: 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-6: 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-7: 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(remainingt0Debt_,
↪   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.t0Debt;
        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: PositionManager & PermitERC721 do not comply with EIP-4494

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/31

Found by

Bauchibred, GimelSec

Summary

The Scope QA explicitly stated an expectation for the contract/code under inspection to comply with EIP-4494. However, the `PermitERC721` and `PositionManager` contracts do not satisfy the requirements of the EIP-4494 standard. Specifically, they lack the implementation of the IERC165 interface and do not indicate support for the 0x5604e225 interface. These discrepancies, mark the contracts as non-compliant with the EIP-4494 standard, which could lead to potential interoperability issues.

Vulnerability Detail

The PermitERC721 and PositionManager contracts have not been implemented according to the specifications of the EIP-4494 standard. They lack the crucial implementation of the IERC165 interface and do not support the 0x5604e225 interface, both of which are mandatory according to the EIP-4494 standard... quoting the EIP: *This EIP requires EIP-165. EIP165 is already required in ERC-721, but is further necessary here in order to register the interface of this EIP. Doing so will allow easy verification if an NFT contract has implemented this EIP or not, enabling them to interact accordingly. The interface of this EIP (as defined in EIP-165) is 0x5604e225. Contracts implementing this EIP MUST have the supportsInterface function return true when called with 0x5604e225.*

Impact

The absence of the IERC165 implementation and lack of support for the 0x5604e225 interface in accordance with the EIP-4494 standard have several potential implications:

1. The `PositionManager` & `PermitERC721` contracts **do not comply with the EIP-4494 standard**.
2. This discrepancy could hamper the contracts' interoperability with other systems and smart contracts, as third-party contracts would be unable to identify their adherence to the `EIP-4494` standard.

Code Snippet

PermitERC721 and PositionManager

Tool used

Manual Audit

Recommendation

To fully comply with the EIP-4494 standard, both the `PermitERC721` and `PositionManager` contracts must implement the IERC165 interface and declare their support for the 0x5604e225 interface. This necessary adjustment will resolve the non-compliance and ensure smooth interoperability with other systems and smart contracts.

Discussion

grandizzy

https://github.com/ajna-finance/contracts/pull/907

dmitriia

> ajna-finance/contracts#907

Looks ok

ctf-sec

Escalate for 10 USDC.

the severity is low, there is no real impact

https://docs.sherlock.xyz/audits/judging/judging

> EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully complying with the EIP it is implementing and there are no adverse effects of this, is considered informational

sherlock-admin

> Escalate for 10 USDC.
>
> the severity is low, there is no real impact
>
> https://docs.sherlock.xyz/audits/judging/judging
>
> > EIP compliance with no integrations: If the protocol does not have external integrations then issues related to code not fully

> complying with the EIP it is implementing and there are no adverse effects of this, is considered informational

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.

Bauchibred

Escalate for 10USDC

While I have great respect for the escalator @ctf-sec's auditing expertise and the due diligence he/she does on findings to escalate where he/she thinks something should be changed, I would have to disagree with this escalation and reiterate that this issue about compliance with 'EIP-4494' is indeed a valid finding which is worthy to be labeled *medium*.

Putting the fact that sponsors confirmed this issue as medium and tagged it a *will fix* aside, below are other factors to support my argument.

Based on the same Sherlock judging criteria link the escalator sent:

> https://docs.sherlock.xyz/audits/judging/judging

The below is the standard Sherlock observes:

> **Hierarchy of truth**: Contest README > Sherlock rules for valid issues > Historical decisions.

And as explained in the above bug report, in fact the sentence under *Summary* reads:

> The Scope's QA explicitly stated an expectation for the contract/code under inspection to comply with EIP-4494.

Meaning that the sponsors assume that their contracts should be in compliance with the EIP and if it's not, auditors are incentivised to submit reports about this and would be paid for this.

Now about the escalator's stand on there being no real impact, I believe is a far reach. If the issue had no real impact, sponsors would not confirm the issue nor would they make a fix to the issue, as has been stated in the above comment. Asides the sponsors' actions, infact even the judge would not submit the issue to sponsors for a review, and as mentioned in the bug report, this obviously hampers the contracts interoperability with other systems and smart contracts, as third-party contracts would be unable to identify their adherence to the EIP-4494 standard.

Additionally, just as another supporting factor to this escalation, though fair enough based on Sherlock's hierarchy of truth *Historical decisions* have the least strength.

Sherlock has previously validated and awarded EIP issues as payment-worthy findings, with one of the famous ones being the `onlyEOA()` modifier from the bluebe rry contest. Key to note that in this particular case no explicit mentions were made by sponsors that they would like to comply with the EIP or even be aware of issues regarding this. Solodit is a now pretty popular tool for auditors and s can be used to find out about other issues regarding EIPs that's been awarded by Sherlock.

I believe this escalation is an important case and the decision going to be made by Sherlock would have to consider the aforementioned points. Also, if Sherlock invalidates this finding and decides that it's not a *payment-worthy* one, then this easily means Sherlock doesn't want auditors to take the contests readMe's Q&A as the optimum truth even when explicit comments have been made about the project's compliance, and want them to know that they would be penalized for submitting a valid issue, which leads to discouragement on the sides of the auditors to even go enough depth to find out about what could happen to the compliance the projects assume they have, and that actually says a bad thing about an audit.

Finally, this is in no way an attempt to have a back and forth with the escalator, and I would accept whatever decision is made by Sherlock on this. But a decision should be made after a consideration of the argument this escalation entails. Thank you.

sherlock-admin

> Escalate for 10USDC
>
> While I have great respect for the escalator @ctf-sec's auditing expertise and the due diligence he/she does on findings to escalate where he/she thinks something should be changed, I would have to disagree with this escalation and reiterate that this issue about compliance with 'EIP-4494' is indeed a valid finding which is worthy to be labeled *medium*.
>
> Putting the fact that sponsors confirmed this issue as medium and tagged it a *will fix* aside, below are other factors to support my argument.
>
> Based on the same Sherlock judging criteria link the escalator sent:
>
> > https://docs.sherlock.xyz/audits/judging/judging
>
> The below is the standard Sherlock observes:
>
> > **Hierarchy of truth**: Contest README > Sherlock rules for valid issues > Historical decisions.
>
> And as explained in the above bug report, in fact the sentence under *Summary* reads:
>
> > The Scope’s QA explicitly stated an expectation for the contract/code under inspection to comply with EIP-4494.
>
> Meaning that the sponsors assume that their contracts should be in compliance with the EIP and if it's not, auditors are incentivised to submit

> reports about this and would be paid for this.
>
> Now about the escalator's stand on there being no real impact, I believe is a far reach. If the issue had no real impact, sponsors would not confirm the issue nor would they make a fix to the issue, as has been stated in the above comment. Asides the sponsors' actions, infact even the judge would not submit the issue to sponsors for a review, and as mentioned in the bug report, this obviously hampers the contracts interoperability with other systems and smart contracts, as third-party contracts would be unable to identify their adherence to the EIP-4494 standard.
>
> Additionally, just as another supporting factor to this escalation, though fair enough based on Sherlock's hierarchy of truth *Historical decisions* have the least strength. Sherlock has previously validated and awarded EIP issues as payment-worthy findings, with one of the famous ones being the `onlyEOA()` modifier from the blueberry contest. Key to note that in this particular case no explicit mentions were made by sponsors that they would like to comply with the EIP or even be aware of issues regarding this. Solodit is a now pretty popular tool for auditors and s can be used to find out about other issues regarding EIPs that's been awarded by Sherlock.
>
> I believe this escalation is an important case and the decision going to be made by Sherlock would have to consider the aforementioned points. Also, if Sherlock invalidates this finding and decides that it's not a *payment-worthy* one, then this easily means Sherlock doesn't want auditors to take the contests readMe's Q&A as the optimum truth even when explicit comments have been made about the project's compliance, and want them to know that they would be penalized for submitting a valid issue, which leads to discouragement on the sides of the auditors to even go enough depth to find out about what could happen to the compliance the projects assume they have, and that actually says a bad thing about an audit.
>
> Finally, this is in no way an attempt to have a back and forth with the escalator, and I would accept whatever decision is made by Sherlock on this. But a decision should be made after a consideration of the argument this escalation entails. Thank you.

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.

0xffff11

Despite the escalation of the second watson being longer and given more arguments. I don't see any intrinsic impact on the issue here. Sponsor confirming does not automatically make the issue a medium. There is definitely a fix to be made. But not adhering to the EIP here does not have enough impact to be considered a medium imo.

dmitriia

https://docs.sherlock.xyz/audits/judging/judging#how-to-identify-a-medium-issue

> How to identify a medium issue: Causes a loss of funds but requires certain external conditions or specific states. Breaks core contract functionality, rendering the contract useless (should not be easily replaced without loss of funds) or leading to unknown potential exploits/loss of funds. Eg: Unable to remove malicious user/collateral from the contract. A material loss of funds, no/minimal profit for the attacker at a considerable cost

Bauchibred

I stand corrected @ctf-sec, @0xffff11 & @dmitriia, I now understand how that is a valid ground to stand on based on sherlock's rules.

MLON33

> > ajna-finance/contracts#907
>
> Looks ok

Moving sign-off by @dmitriia here for clarity.

Issue M-2: 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

Bauchibred, 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.

Issue M-3: 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-4: 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-5: 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

SHERLOCK

Issue M-6: Mathematical Discrepancies in equations used for calculating Interest Rates

Source: https://github.com/sherlock-audit/2023-04-ajna-judging/issues/104

Found by

Chinmay

Summary

The two equations representing MAU - TU relationships that are used to check whether the Interest Rate should be decreased or Increased based on current state, are not homogenous. The implementation of these yields different values in calculations.

Vulnerability Detail

The two equations seen at `PoolCommons.sol:L294` and `L297` are checks that allow increasing or decreasing interest rates based on the current values of `Meaningful Actual Utilization MAU` and `Target Utilization TU`. The two equations are comparing same quantities mau and tu and should have same similar scaling/downscaling. Here is the code :

```
    if (4 * (tu - mau102) < (((tu + mau102 - 1e18) / 1e9) ** 2) - 1e18) {
        newInterestRate_ = Maths.wmul(poolState_.rate, INCREASE_COEFFICIENT);
    // decrease rates if 4*(tu-mau) > 1-(tu+mau-1)^2
    } else if (4 * (tu - mau) > 1e18 - ((tu + mau - 1e18) ** 2) / 1e18) {
        newInterestRate_ = Maths.wmul(poolState_.rate, DECREASE_COEFFICIENT);
    }

    // bound rates between 10 bps and 50000%
    newInterestRate_ = Maths.min(500 * 1e18, Maths.max(0.001 * 1e18,
 ↪  newInterestRate_));
}
```

We notice a difference in how they use downscaling by 1e9 or 1e18 to attain WAD(1e18) precision on both sides of the inequality.

We are interested in the term `(tu + mau102 - 1e18) / 1e9) ** 2)`. For the increase rate check, `(tu + mau102 - 1e18) / 1e9) ** 2)` is used, but notice that for the decrease rate check, `((tu + mau - 1e18) ** 2) / 1e18)` is used instead. This discrepancy will lead to a different set of results for this expression in some cases.

When I asked the developers, why this discrepancy exists, they said `"so the thing is that during the invariant testing, with some huge values, the first one` `overflowed, let me find the commit so we decided to make it allow higher values, by dividing by 1e9 and then pow 2, instead having the 1e18 at pow 2 and then / 1e18. But the else branch wasn't changed, as we hit no failure there."`

They changed the first equation to deal with an overflow issue, but they did not consider changing the second one because it did not overflow. But they expected it to be mathematically the same. One of the developers said, `"mathematically they should be the same IMO... not sure why differences, granted the else clause would be better written same way e.g. (4 * (tu - mau) > 1e18 - ((tu + mau - 1e18) / 1e9) ** 2)"`

The effect of this discrepancy is that for some sets of values both these terms aren't equivalent. I plotted these two expressions here : https://www.desmos.com/calculator/cjaiyhtmob So the mathematical equation yields different values than was expected and what the equation should represent.

Impact

The mathematical equation used was not the one that was intended. There is a clear discrepancy in the results of these equations and thus this is a medium severity issue because the protocol will not work as expected in some cases.

One of the developers agreed, `"mathematically they should be the same IMO... not sure why differences, granted the else clause would be better written same way e.g. (4 * (tu - mau) > 1e18 - ((tu + mau - 1e18) / 1e9) ** 2)"`

Code Snippet

https://github.com/sherlock-audit/2023-04-ajna/blob/e2439305cc093204a0d927aac19d898f4a0edb3d/ajna-core/src/libraries/external/PoolCommons.sol#L297

Tool used

Manual Review

Recommendation

Change Line 297 from `4 * (tu - mau) > 1e18 - ((tu + mau - 1e18) ** 2) / 1e18)` to `4 * (tu - mau) > 1e18 - ((tu + mau - 1e18) / 1e9) ** 2)`

Discussion

grandizzy

We acknowledge this is a discrepancy in our code and will be fixed in https://github.com/ajna-finance/contracts/pull/903

dmitriia

> We acknowledge this is a discrepancy in our code and will be fixed in ajna-finance/contracts#903

Fix looks ok

ctf-sec

Escalate for 10 USDC.

the severity is low unless the report describe what is the true impact of the mismatched implementation

sherlock-admin

> Escalate for 10 USDC.
>
> the severity is low unless the report describe what is the true impact of the mismatched implementation

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

Escalate for 10 USDC I'd like to elaborate the impact to justify medium severity

1. The two expressions above are not same for some sets of values as shown in the graph. This means that intended implementation is different from actual code. Not working as intended is itself a criteria based on previous judged issues. The protocol wanted to increase/decrease rates based on these conditions, but now the decrease rates conditions may result in an incorrect decrease operation when it should have behaved homogenous to the increase rates condition check.
2. Issues like this where mathematical implementations differ from desired/documented one are considered a medium severity issue : https://github.com/code-423n4/2023-03-mute-findings/issues/19
3. This will prevent interest rate from decreasing when it should have decreased (and the other way round), which means the new Interest rate may be wrong and this value is then propagated to the state (for example at PoolCommons#L181 new Interest rate is not updated if it was the same as before, and the mentioned discrepancy can result in this because it was incorrect to not change it when it should have) and used in the system. What this means is an incorrect value of interest rate is being used across the system, which leads to it being a medium severity issue.

sherlock-admin

> Escalate for 10 USDC I'd like to elaborate the impact to justify medium severity
>
> 1. The two expressions above are not same for some sets of values as shown in the graph. This means that intended implementation is different from actual code. Not working as intended is itself a criteria based on previous judged issues. The protocol wanted to increase/decrease rates based on these conditions, but now the decrease rates conditions may result in an incorrect decrease operation when it should have behaved homogenous to the increase rates condition check.
> 2. Issues like this where mathematical implementations differ from desired/documented one are considered a medium severity issue : https://github.com/code-423n4/2023-03-mute-findings/issues/19
> 3. This will prevent interest rate from decreasing when it should have decreased (and the other way round), which means the new Interest rate may be wrong and this value is then propagated to the state (for example at PoolCommons#L181 new Interest rate is not updated if it was the same as before, and the mentioned discrepancy can result in this because it was incorrect to not change it when it should have) and used in the system. What this means is an incorrect value of interest rate is being used across the system, which leads to it being a medium severity issue.

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.

0xffff11

There is no real impact shown on the issue. Discrepancies code-docs if there is no loss of funds is not a medium. In this case it is not demonstrated that having different equations affect the protocol. I would agree at the current state with first escalation if no extra info is given, low

dmitriia

Tend to agree, it looks like valid low.

The difference with (2) is that there is no clear material loss path here, such as stakers losing the rewards there:

> As we will see this means that all stakers that do not have a tokenRatio of exactly equal 0 or exactly equal 1 lose out on rewards that they should

> receive according to the documentation.

chinmay-farkya

It does affect the protocol imo. See point 3 in above escalation. Interest rates not being decreased when they should have is incorrect logic. I will reiterate the third point here,

"This will prevent interest rate from decreasing when it should have decreased (and the other way round), which means the new Interest rate may be wrong and this value is then propagated to the state (for example at PoolCommons#L181 new Interest rate is not updated if it was the same as before, and the mentioned discrepancy can result in this because it was incorrect to not change it when it should have) and used in the system. What this means is an incorrect value of interest rate is being used across the system, which leads to it being a medium severity issue."

This interest rate is used to calculate interests which means sometimes the borrower may be charged more than they deserve and sometimes less than they deserve, which leads to loss of funds for the interest accruals as well as the borrower.

MLON33

> > We acknowledge this is a discrepancy in our code and will be fixed in ajna-finance/contracts#903
>
> Fix looks ok

Moving sign-off by @dmitriia here for clarity.

Issue M-7: 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.