SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Ajna
Prepared by: Sherlock
Lead Security Expert: hyh
Dates Audited: January 9 - January 30, 2023
Prepared on: February 13, 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

- ./contracts/src files and any files they import
- ./ecosystem-coordination/src files and any files they import

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
21	12

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
Jeiwan
berndartmueller
ctf_sec
koxuan
Blockian
MalfurionWhitehat
yixxas
oxcm
CRYP70
cducrest-brainbot
Chinmay
peanuts
minhtrng

Issue H-1: RewardsManager doesn't delete old bucket snapshot info on unstaking

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/183

Found by

hyh

Summary

RewardsManager's unstake() use `delete stakes[tokenId_]` to clear old stake state, but `snapshot` is the nested mapping in the `StakeInfo` structure and will not be reset this way as delete operation do not traverse through nested mappings as it lacks key set information.

Vulnerability Detail

`stakes[tokenId_]` gets written on staking and `mapping(uint256 => BucketState) snapshot` is written for the *current* list of buckets. This means if this list persists and there were no bucket changes it's ok as new values will be overwritten on next stake.

But, if Bob the staker has changed his composition of buckets and his second stake takes place over another set, possibly intersecting with the first one, old part will persist. If then Bob's `positionIndexes = positionManager.getPositionIndexes(tokenId_)` changed after the second stake, say as a result of PositionManager's moveLiquidity(), and indices from the first set were added there, their snapshot values from the first stake will be reused.

Impact

If Bob knows this it will be straightforward for him to exploit the mechanics, obtaining extra rewards (interest earned will be counted from the first stake time for old positions) at the expense of other stakers.

Code Snippet

RewardsManager's unstake() deletes `stakes[tokenId_]`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L187-L203

```
function unstake(
    uint256 tokenId_
) external override {
```


SHERLOCK

```
    if (msg.sender != stakes[tokenId_].owner) revert NotOwnerOfDeposit();

    address ajnaPool = stakes[tokenId_].ajnaPool;

    // claim rewards, if any
    _claimRewards(tokenId_, IPool(ajnaPool).currentBurnEpoch());

    delete stakes[tokenId_];

    emit Unstake(msg.sender, ajnaPool, tokenId_);

    // transfer LP NFT from contract to sender
    IERC721(address(positionManager)).safeTransferFrom(address(this),
↪   msg.sender, tokenId_);
}
```

`stakes[tokenId_]` is `StakeInfo` structure:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L76

```
mapping(uint256 => StakeInfo) internal stakes;  // tokenID => Stake info
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L21

```
import { StakeInfo, BucketState } from
↪   './interfaces/rewards/IRewardsManagerState.sol';
```

It contains `snapshot` mapping elemewnt that will not be cleared on delete:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/interfaces/rewards/IRewardsManagerState.sol#L56-L62

```
struct StakeInfo {
    address ajnaPool;                         // address of the Ajna pool the
↪   NFT corresponds to
    uint96  lastInteractionBurnEpoch;         // last burn event the stake
↪   interacted with the rewards contract
    address owner;                            // owner of the LP NFT
    uint96  stakingEpoch;                     // epoch at staking time
    mapping(uint256 => BucketState) snapshot; // the LP NFT's balances and
↪   exchange rates in each bucket at the time of staking
}
```

Per operation docs:

https://docs.soliditylang.org/en/latest/types.html#delete

So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings

This way restaking the `tokenId_` will reuse the old `snapshot` mapping.

`BucketState` structure consists of `rateAtStakeTime` and `lpsAtStakeTime`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/interfaces/rewards/IRewardsManagerState.sol#L64-L67

```
struct BucketState {
    uint256 lpsAtStakeTime;  // [RAY] LP amount the NFT owner is entitled in
↪   current bucket at the time of staking
    uint256 rateAtStakeTime; // [RAY] current bucket exchange rate at the time
↪   of staking (RAY)
}
```

Both are written on staking, but only for the list of indices as of time of staking:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L144-L162

```
uint256[] memory positionIndexes = positionManager.getPositionIndexes(tokenId_);

for (uint256 i = 0; i < positionIndexes.length; ) {

    uint256 bucketId = positionIndexes[i];

    BucketState storage bucketState = stakeInfo.snapshot[bucketId];

    // record the number of lp tokens in bucket at the time of staking
    bucketState.lpsAtStakeTime = positionManager.getLPTokens(
        tokenId_,
        bucketId
    );
    // record the bucket exchange rate at the time of staking
    bucketState.rateAtStakeTime = IPool(ajnaPool).bucketExchangeRate(bucketId);

    // iterations are bounded by array length (which is itself bounded),
↪   preventing overflow / underflow
    unchecked { ++i; }
}
```

`rateAtStakeTime` and `lpsAtStakeTime` are used for the accrued interest calculation in the _calculateNextEpochRewards():

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L333-L351

```
uint256 bucketRate;
if (epoch_ != stakingEpoch_) {

    // if staked in a previous epoch then use the initial exchange rate of epoch
    bucketRate = bucketExchangeRates[ajnaPool_][bucketIndex][epoch_];
} else {

    // if staked during the epoch then use the bucket rate at the time of staking
    bucketRate = bucketSnapshot.rateAtStakeTime;
}

// calculate the amount of interest accrued in current epoch
uint256 interestEarned = _calculateExchangeRateInterestEarned(
    ajnaPool_,
    nextEpoch,
    bucketIndex,
    bucketSnapshot.lpsAtStakeTime,
    bucketRate
);
```

This happens for the current `positionIndexes = positionManager.getPositionIndexes(tokenId_)`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L272-L292

```
function _calculateAndClaimRewards(
    uint256 tokenId_,
    uint256 epochToClaim_
) internal returns (uint256 rewards_) {

    address ajnaPool       = stakes[tokenId_].ajnaPool;
    uint256 lastBurnEpoch = stakes[tokenId_].lastInteractionBurnEpoch;
    uint256 stakingEpoch   = stakes[tokenId_].stakingEpoch;

    uint256[] memory positionIndexes =
↪   positionManager.getPositionIndexes(tokenId_);

    // iterate through all burn periods to calculate and claim rewards
    for (uint256 epoch = lastBurnEpoch; epoch < epochToClaim_; ) {

        uint256 nextEpochRewards = _calculateNextEpochRewards(
            tokenId_,
            epoch,
            stakingEpoch,
            ajnaPool,
            positionIndexes
```

```
        );
```

Say Bob restaked, the `snapshot` persisted. Then if positions changed since the second stake and new indices have been there before (i.e. old ones were *readded*, so they weren't reset on the second stake() as were added later, but their values end up not being void as they were there on the first stake and persisted), then their values will be reused from the first Bob's staking.

This will expand Bob's interest earned reading:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L368-L398

```
/**
 *  @notice Calculate the amount of interest that has accrued to a lender in a
 ↪  bucket based upon their LPs.
 *  @param  pool_           Address of the pool whose exchange rates are being
 ↪  checked.
 *  @param  nextEventEpoch_ The next event epoch to check the exchange rate for.
 *  @param  bucketIndex_    Index of the bucket to check the exchange rate for.
 *  @param  bucketLPs       Amount of LPs in bucket.
 *  @param  exchangeRate_   Exchange rate in current epoch.
 *  @return interestEarned_ The amount of interest accrued.
 */
function _calculateExchangeRateInterestEarned(
    address pool_,
    uint256 nextEventEpoch_,
    uint256 bucketIndex_,
    uint256 bucketLPs,
    uint256 exchangeRate_
) internal view returns (uint256 interestEarned_) {

    if (exchangeRate_ != 0) {

        uint256 nextExchangeRate =
 ↪  bucketExchangeRates[pool_][bucketIndex_][nextEventEpoch_];

        // calculate interest earned only if next exchange rate is higher than
 ↪  current exchange rate
        if (nextExchangeRate > exchangeRate_) {

            // calculate the equivalent amount of quote tokens given the stakes
 ↪  lp balance,
            // and the exchange rate at the next and current burn events
            interestEarned_ = Maths.rayToWad(Maths.rmul(nextExchangeRate -
 ↪  exchangeRate_, bucketLPs));
        }
```

```
    }
}
```

Tool used

Manual Review

Recommendation

IRewardsManagerState.BucketState doesn't contain any nested structures, so `delete bucketState` will reset it fully:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/interfaces/rewards/IRewardsManagerState.sol#L64-L67

```
struct BucketState {
    uint256 lpsAtStakeTime;  // [RAY] LP amount the NFT owner is entitled in
↪   current bucket at the time of staking
    uint256 rateAtStakeTime; // [RAY] current bucket exchange rate at the time
↪   of staking (RAY)
}
```

Consider clearing the current stake snapshots on unstaking, for example:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/RewardsManager.sol#L187-L203

```
    function unstake(
        uint256 tokenId_
    ) external override {
        if (msg.sender != stakes[tokenId_].owner) revert NotOwnerOfDeposit();

        address ajnaPool = stakes[tokenId_].ajnaPool;

        // claim rewards, if any
        _claimRewards(tokenId_, IPool(ajnaPool).currentBurnEpoch());

+       uint256[] memory positionIndexes =
↪   positionManager.getPositionIndexes(tokenId_);
+       for (uint256 i = 0; i < positionIndexes.length; ) {
+           delete stakeInfo.snapshot[positionIndexes[i]]; // BucketState
+           unchecked { ++i; }
+       }
        delete stakes[tokenId_];

        emit Unstake(msg.sender, ajnaPool, tokenId_);
```

```
        // transfer LP NFT from contract to sender
        IERC721(address(positionManager)).safeTransferFrom(address(this),
↪   msg.sender, tokenId_);
    }
```

Issue H-2: Permanent freezing of unclaimed yield

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/151

Found by

Blockian

Summary

A user can accidentally freeze potential rewards from the `RewardsManager.sol`

Vulnerability Detail

On `claimRewards` in `RewardsManager` contract the `epochToClaim_` is not limited to the `currentBurnEpoch()`, thus allowing a user to send `epochToClaim_ > currentBurnEpoch`, rendering `isEpochClaimed[tokenId_][epochToClaim_]` true from all epochs from `0 - epochToClaim_` and disallowing the user rewards he may have received in future epochs

epochToClaim_ no limit check

Impact

By accidentally sending the wrong epoch a user may freeze unclaimed yield with no way of getting it back

Code Snippet

```
function claimRewards(
    uint256 tokenId_,
    uint256 epochToClaim_ // not limited
) external override {
    if (msg.sender != stakes[tokenId_].owner) revert NotOwnerOfDeposit();

    if (isEpochClaimed[tokenId_][epochToClaim_]) revert AlreadyClaimed(); //
 ↪  will revert for epochs the user didn't receive the rewards from yet

    _claimRewards(tokenId_, epochToClaim_);
}
```

POC

Add this test to the `RewardsManager.t.sol`

```
function testClaimRewardsFroozeUnclaimedYield() external {
    skip(10);

    uint256[] memory depositIndexes = new uint256[](5);
    depositIndexes[0] = 9;
    depositIndexes[1] = 1;
    depositIndexes[2] = 2;
    depositIndexes[3] = 3;
    depositIndexes[4] = 4;
    MintAndMemorializeParams memory mintMemorializeParams =
 ↪  MintAndMemorializeParams({
        indexes: depositIndexes,
        minter: _minterOne,
        mintAmount: 1000 * 1e18,
        pool: _poolOne
    });

    uint256 tokenIdOne = _mintAndMemorializePositionNFT(mintMemorializeParams);
    _stakeToken(address(_poolOne), _minterOne, tokenIdOne);

    uint256 currentBurnEpoch = _poolOne.currentBurnEpoch();

    changePrank(_minterOne);
    _rewardsManager.claimRewards(tokenIdOne, currentBurnEpoch + 10);

    for (uint i = 1; i <= 10; i++) {
        vm.expectRevert(IRewardsManagerErrors.AlreadyClaimed.selector);
        _rewardsManager.claimRewards(tokenIdOne, currentBurnEpoch + i); // the
 ↪  user got all 10 next burn epochs rewards frozen as he receives
 ↪  AlreadyClaimed for all of them
    }
}
```

Tool used

Manual Review

Recommendation

There are 2 main ways to fix this issue. Either limit the `for` loops that depend on `epochToClaim_` for example instead of

```
for (uint256 epoch = lastBurnEpoch; epoch < epochToClaim_; )
```

use

```
uint256 curBurnEpoch = IPool(ajnaPool).currentBurnEpoch();
uint256 maxEpoch = epochToClaim_ > curBurnEpoch ? curBurnEpoch : epochToClaim_;
for (uint256 epoch = lastBurnEpoch; epoch < maxEpoch; )
```

Which is a BAD solution (explanation in detail in the next issue)

Or simply add a require statement when calling `claimRewards`

```
function claimRewards(
    uint256 tokenId_,
    uint256 epochToClaim_
) external override {
    if (epochToClaim_ > IPool(stakes[tokenId_].ajnaPool).currentBurnEpoch())
↪   revert EpochNotAvailableYet();

    // rest of the function
}
```

Which is a better solution

Issue H-3: Anyone who approved quote tokens to a pool can be forced to take

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/145

Found by

Jeiwan

Summary

Taking may be executed on behalf of any address who approved spending of quote tokens to a pool: such address will pay quote tokens and will receive collateral.

Vulnerability Detail

ERC20Pool and ERC721Pool implement the `take` functions, which buy collateral from auction in exchange for quote tokens. The address to pull quote tokens from is specified in the `callee_` argument, which allows anyone to call the functions and pass an address that has previously approved spending of the quote token to the pool. As a result, such an address will pay for the liquidation and will receive the collateral.

Impact

Anyone can initiate a take on behalf of another user. Such user can be a lender who has previously approved spending of the quote token to the pool. Calling `take` with the user's address specified as the `callee_` argument will result in:

1. the user receiving collateral, which may have low value;
2. the user paying the quote token to repay the debt being taken.

Code Snippet

ERC20Pool.sol#L460 ERC721Pool.sol#L463

Tool used

Manual Review

Recommendation

In the `ERC20Pool.take` and `ERC721Pool.take` functions, consider transferring collateral only from `msg.sender`. Alternatively, consider checking that `callee_` has approved spending quote tokens to `msg.sender`.

Issue H-4: CryptoPunks NFTs may be stolen via deposit frontrunning

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/140

Found by

Jeiwan

Summary

Depositing of CryptoPunks NFTs may be front run, a malicious actor may deposit someone else's CryptoPunks NFT.

Vulnerability Detail

Due to the CryptoPunks NFT collection not implementing the ERC721 standard, depositing of CryptoPunks NFTs is implemented via a direct sale:

1. token owner needs to call offerPunkForSaleToAddress and set the `toAddress` value to the address of the pool the token will be deposited to;
2. token owner then calls the addCollateral function of the ERC721 pool;
3. the pool buys the token from its owner.

However, `addCollateral` can be called by anyone: the pool will buy the token and will deposit it on the caller's account even if the caller is not the owner of the token.

Impact

CryptoPunks NFTs owner may lose their NFTs when trying to deposit them to an ERC721 pool. A malicious actor may front run the depositing and deposit the NFTs to their account. The malicious actor may then withdraw the NFTs.

Code Snippet

ERC721Pool.sol#L577 CryptoPunksMarket:

```
function offerPunkForSaleToAddress(uint punkIndex, uint minSalePriceInWei,
↪   address toAddress) {
    if (!allPunksAssigned) throw;
    if (punkIndexToAddress[punkIndex] != msg.sender) throw;
    if (punkIndex >= 10000) throw;
    punksOfferedForSale[punkIndex] = Offer(true, punkIndex, msg.sender,
↪   minSalePriceInWei, toAddress);
    PunkOffered(punkIndex, minSalePriceInWei, toAddress);
```

```
}

function buyPunk(uint punkIndex) payable {
    if (!allPunksAssigned) throw;
    Offer offer = punksOfferedForSale[punkIndex];
    if (punkIndex >= 10000) throw;
    if (!offer.isForSale) throw;                // punk not actually for sale
    if (offer.onlySellTo != 0x0 && offer.onlySellTo != msg.sender) throw;  //
↪   punk not supposed to be sold to this user
    if (msg.value < offer.minValue) throw;      // Didn't send enough ETH
    if (offer.seller != punkIndexToAddress[punkIndex]) throw; // Seller no
↪   longer owner of punk

    address seller = offer.seller;

    punkIndexToAddress[punkIndex] = msg.sender;
    balanceOf[seller]--;
    balanceOf[msg.sender]++;
    Transfer(seller, msg.sender, 1);

    punkNoLongerForSale(punkIndex);
    pendingWithdrawals[seller] += msg.value;
    PunkBought(punkIndex, msg.value, seller, msg.sender);

    // Check for the case where there is a bid from the new owner and refund it.
    // Any other bid can stay in place.
    Bid bid = punkBids[punkIndex];
    if (bid.bidder == msg.sender) {
        // Kill bid and refund value
        pendingWithdrawals[msg.sender] += bid.value;
        punkBids[punkIndex] = Bid(false, punkIndex, 0x0, 0);
    }
}
```

Tool used

Manual Review

Recommendation

Before buying a CryptoPunks NFT, consider checking that `msg.sender` is the owner of the token. For example:

```
diff --git a/contracts/src/ERC721Pool.sol b/contracts/src/ERC721Pool.sol
index b1bf36b..a512a9d 100644
--- a/contracts/src/ERC721Pool.sol
+++ b/contracts/src/ERC721Pool.sol
```

```
@@ -574,6 +574,7 @@ contract ERC721Pool is FlashloanablePool, IERC721Pool {

↪   ICryptoKitties(_getArgAddress(COLLATERAL_ADDRESS)).transferFrom(msg.sender
↪   ,address(this), tokenId);
             }
             else{
+                    require(ICryptoPunks(_getArgAddress(COLLATERAL_ADDRESS)).punkIn
↪   dexToAddress(tokenId) == msg.sender);

↪   ICryptoPunks(_getArgAddress(COLLATERAL_ADDRESS)).buyPunk(tokenId);
             }
```

Discussion

grandizzy

will fix with the fix for https://github.com/sherlock-audit/2023-01-ajna-judging/issues/163

Issue H-5: scaledQuoteTokenAmount isn't updated to be collateral sell value in the quote token constraint case of _calculateTakeFlowsAndBondChange

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/139

Found by

hyh

Summary

`scaledQuoteTokenAmount` isn't `C * p`, but `C * p * (1 - BFP)` for quote token amount constraint case of _calculateTakeFlowsAndBondChange().

Vulnerability Detail

First case of the _calculateTakeFlowsAndBondChange() logic needs to use `scaledQuoteTokenAmount` in two steps, first as a constraint, then as a total collateral value. The second update is now missed. It affects kicker's reward as the difference takes place when `borrowerPrice < auctionPrice`, i.e. when `vars.isRewarded` is true.

Impact

`scaledQuoteTokenAmount` is then used for kicker's bond change calculation, so in the quote token constraint case kickers will have the reward based on `C * p * (1 - BFP)`. As this value is proportional to `BFP`, the higher the reward should be, the more incorrect it will be, i.e. `(1 - BFP) * BFP` instead of `BFP`.

As this is regular functionality, there is no low probability prerequisites, and kicker's reward loss is material, setting the severity to be high.

Code Snippet

_calculateTakeFlowsAndBondChange() has first logic branch where quote token used to purchase is a constraint:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L1139-L1149

```
vars.scaledQuoteTokenAmount = (vars.unscaledDeposit != type(uint256).max) ?
↪  Maths.wmul(vars.unscaledDeposit, vars.bucketScale) : type(uint256).max;

uint256 borrowerCollateralValue = Maths.wmul(totalCollateral_, borrowerPrice);
```


SHERLOCK

```
if (vars.scaledQuoteTokenAmount <= vars.borrowerDebt &&
↪ vars.scaledQuoteTokenAmount <= borrowerCollateralValue) {
    // quote token used to purchase is constraining factor
    vars.collateralAmount         =
↪   _roundToScale(Maths.wdiv(vars.scaledQuoteTokenAmount, borrowerPrice),
↪   collateralScale_);
    vars.t0RepayAmount            = Maths.wdiv(vars.scaledQuoteTokenAmount,
↪   inflator_);
    vars.unscaledQuoteTokenAmount = vars.unscaledDeposit;

}
```

`vars.scaledQuoteTokenAmount` is used for the `bondChange` calculation and per documentation has to be equal to `C * p`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L1166-L1172

```
if (vars.isRewarded) {
    // take is above neutralPrice, Kicker is rewarded
    vars.bondChange = Maths.wmul(vars.scaledQuoteTokenAmount, uint256(vars.bpf));
} else {
    // take is above neutralPrice, Kicker is penalized
    vars.bondChange = Maths.wmul(vars.scaledQuoteTokenAmount,
↪   uint256(-vars.bpf));
}
```

And it is `vars.scaledQuoteTokenAmount = CollateralAmount * Price` in 2nd and 3rd cases, not not in 1st:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L1124-L1175

```
function _calculateTakeFlowsAndBondChange(
    uint256              totalCollateral_,
    uint256              inflator_,
    uint256              collateralScale_,
    TakeLocalVars memory vars
) internal pure returns (
    TakeLocalVars memory
) {
    // price is the current auction price, which is the price paid by the LENDER
↪   for collateral
    // from the borrower point of view, the price is actually (1-bpf) * price,
↪   as the rewards to the
    // bond holder are effectively paid for by the borrower.
    uint256 borrowerPayoffFactor = (vars.isRewarded) ? Maths.WAD -
↪   uint256(vars.bpf)                       : Maths.WAD;
```

```
    uint256 borrowerPrice         = (vars.isRewarded) ?
↪   Maths.wmul(borrowerPayoffFactor, vars.auctionPrice) : vars.auctionPrice;

    // If there is no unscaled quote token bound, then we pass in max, but that
↪   cannot be scaled without an overflow.  So we check in the line below.
    vars.scaledQuoteTokenAmount = (vars.unscaledDeposit != type(uint256).max) ?
↪   Maths.wmul(vars.unscaledDeposit, vars.bucketScale) : type(uint256).max;

    uint256 borrowerCollateralValue = Maths.wmul(totalCollateral_,
↪   borrowerPrice);

    if (vars.scaledQuoteTokenAmount <= vars.borrowerDebt &&
↪   vars.scaledQuoteTokenAmount <= borrowerCollateralValue) {
        // quote token used to purchase is constraining factor
        vars.collateralAmount         =
↪   _roundToScale(Maths.wdiv(vars.scaledQuoteTokenAmount, borrowerPrice),
↪   collateralScale_);
        vars.t0RepayAmount            = Maths.wdiv(vars.scaledQuoteTokenAmount,
↪   inflator_);
        vars.unscaledQuoteTokenAmount = vars.unscaledDeposit;

    } else if (vars.borrowerDebt <= borrowerCollateralValue) {
        // borrower debt is constraining factor
        vars.collateralAmount         =
↪   _roundToScale(Maths.wdiv(vars.borrowerDebt, borrowerPrice),
↪   collateralScale_);
        vars.t0RepayAmount            = vars.t0Debt;
        vars.unscaledQuoteTokenAmount = Maths.wdiv(vars.borrowerDebt,
↪   vars.bucketScale);

        vars.scaledQuoteTokenAmount   = (vars.isRewarded) ?
↪   Maths.wdiv(vars.borrowerDebt, borrowerPayoffFactor) : vars.borrowerDebt;

    } else {
        // collateral available is constraint
        vars.collateralAmount         = totalCollateral_;
        vars.t0RepayAmount            = Maths.wdiv(borrowerCollateralValue,
↪   inflator_);
        vars.unscaledQuoteTokenAmount = Maths.wdiv(borrowerCollateralValue,
↪   vars.bucketScale);

        vars.scaledQuoteTokenAmount   = Maths.wmul(vars.collateralAmount,
↪   vars.auctionPrice);
    }

    if (vars.isRewarded) {
        // take is above neutralPrice, Kicker is rewarded
```

```
        vars.bondChange = Maths.wmul(vars.scaledQuoteTokenAmount,
↪   uint256(vars.bpf));
    } else {
        // take is above neutralPrice, Kicker is penalized
        vars.bondChange = Maths.wmul(vars.scaledQuoteTokenAmount,
↪   uint256(-vars.bpf));
    }

    return vars;
}
```

`vars.scaledQuoteTokenAmount` in the first case has dual role, at first it is a constraint, then it is `C * p` computation base value, so it is to be used iteratively, first as a constraint, then updated to be `CollateralAmount * Price`.

Tool used

Manual Review

Recommendation

Consider setting the `scaledQuoteTokenAmount` to the `C * p` as a final step of quote token amount constraint case:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L1139-L1149

```
        vars.scaledQuoteTokenAmount = (vars.unscaledDeposit !=
↪   type(uint256).max) ? Maths.wmul(vars.unscaledDeposit, vars.bucketScale) :
↪   type(uint256).max;

        uint256 borrowerCollateralValue = Maths.wmul(totalCollateral_,
↪   borrowerPrice);

        if (vars.scaledQuoteTokenAmount <= vars.borrowerDebt &&
↪   vars.scaledQuoteTokenAmount <= borrowerCollateralValue) {
            // quote token used to purchase is constraining factor
            vars.collateralAmount         =
↪   _roundToScale(Maths.wdiv(vars.scaledQuoteTokenAmount, borrowerPrice),
↪   collateralScale_);
            vars.t0RepayAmount            =
↪   Maths.wdiv(vars.scaledQuoteTokenAmount, inflator_);
            vars.unscaledQuoteTokenAmount = vars.unscaledDeposit;
+           vars.scaledQuoteTokenAmount   = Maths.wmul(vars.collateralAmount,
↪   vars.auctionPrice);
        }
```

Issue H-6: removeCollateral miss bankrupcy logic and can make future LPs sharing losses with the current ones

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/133

Found by

hyh, Jeiwan, peanuts, yixxas

Summary

LenderActions' removeCollateral() do not checks for bucket solvency after it has removed a collateral from there. This can lead to losses for future depositors of the bucket.

Vulnerability Detail

Bankrupcy check logic now exist in all asset removing functions. That prevent a situation when a bucket defaults, but next LP deposit makes in solvent again and next LP shared losses with the old ones this way without having such intent.

For example, mergeOrRemoveCollateral() calls _removeMaxCollateral() that do check affected bucket for bankrupcy. removeCollateral() do not check for that despite insolvency situation for a bucket can occur after collateral was removed.

Impact

When bucket defaults, but no bankrupcy is checked and no such flag is set, the next LP depositors have to bail out previous, i.e. have to share their losses.

That's a loss for next LPs by unconditional transfer from them to the previous ones.

As removeCollateral() is a part of base functionality that to be used frequently and bucket defaults can routinely happen, so there is no low probability prerequisites, and given the loss for future bucket depositors, setting the severity to be high.

Code Snippet

There is no bucket bankrupcy logic in removeCollateral(), i.e. when there is no quote tokens in the bucket, `lpAmount_ < bucketLPs`, but `bucketCollateral <= collateralAmount_`, bucket de facto defaults, but no such flag is set:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/LenderActions.sol#L379-L414

```
function removeCollateral(
    ...
) external returns (uint256 lpAmount_) {
    ...

    Lender storage lender = bucket.lenders[msg.sender];

    uint256 lenderLpBalance;
    if (bucket.bankruptcyTime < lender.depositTime) lenderLpBalance = lender.lps;
    if (lenderLpBalance == 0 || lpAmount_ > lenderLpBalance) revert
↪   InsufficientLPs();

    // update lender LPs balance
    lender.lps -= lpAmount_;

    // update bucket LPs and collateral balance
    bucket.lps        -= Maths.min(bucketLPs, lpAmount_);
    bucket.collateral -= Maths.min(bucketCollateral, amount_);
}
```

The check is present in _removeMaxCollateral():

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/LenderActions.sol#L619-L630

```
    // update bucket LPs and collateral balance
    bucketLPs         -= Maths.min(bucketLPs, lpAmount_);
    bucketCollateral  -= Maths.min(bucketCollateral, collateralAmount_);
    bucket.collateral  = bucketCollateral;
    if (bucketCollateral == 0 && bucketDeposit == 0 && bucketLPs != 0) {
        emit BucketBankruptcy(index_, bucketLPs);
        bucket.lps            = 0;
        bucket.bankruptcyTime = block.timestamp;
    } else {
        bucket.lps = bucketLPs;
    }
}
```

And removeQuoteToken():

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/LenderActions.sol#L310-L368

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


SHERLOCK

```
    RemoveQuoteParams calldata params_
) external returns (uint256 removedAmount_, uint256 redeemedLPs_, uint256 lup_) {
    ...

    // update lender and bucket LPs balances
    lender.lps -= redeemedLPs_;

    uint256 lpsRemaining = removeParams.bucketLPs - redeemedLPs_;

    if (removeParams.bucketCollateral == 0 && unscaledRemaining == 0 &&
↪   lpsRemaining != 0) {
        emit BucketBankruptcy(params_.index, lpsRemaining);
        bucket.lps            = 0;
        bucket.bankruptcyTime = block.timestamp;
    } else {
        bucket.lps = lpsRemaining;
    }

    emit RemoveQuoteToken(msg.sender, params_.index, removedAmount_,
↪   redeemedLPs_, lup_);
}
```

Tool used

Manual Review

Recommendation

Consider adding the bankruptcy check similarly to other asset removal functions:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/LenderActions.sol#L379-L414

```
    function removeCollateral(
        ...
    ) external returns (uint256 lpAmount_) {
        ...

        Lender storage lender = bucket.lenders[msg.sender];

        uint256 lenderLpBalance;
        if (bucket.bankruptcyTime < lender.depositTime) lenderLpBalance =
↪   lender.lps;
        if (lenderLpBalance == 0 || lpAmount_ > lenderLpBalance) revert
↪   InsufficientLPs();

        // update lender LPs balance
```

```
        lender.lps -= lpAmount_;

        // update bucket LPs and collateral balance
-       bucket.lps        -= Maths.min(bucketLPs, lpAmount_);
-       bucket.collateral -= Maths.min(bucketCollateral, amount_);
+       uint256 bucketLPs = bucket.lps - Maths.min(bucketLPs, lpAmount_);
+       uint256 bucketCollateral = bucket.collateral -
↪   Maths.min(bucketCollateral, amount_);
+       uint256 bucketDeposit = Deposits.valueAt(deposits_, index_);

+       if (bucketCollateral == 0 && bucketDeposit == 0 && bucketLPs != 0) {
+           emit BucketBankruptcy(index_, bucketLPs);
+           bucket.lps = 0;
+           bucket.bankruptcyTime = block.timestamp;
+       } else {
+           bucket.lps = bucketLPs;
+       }
+       bucket.collateral = bucketCollateral;
+   }
```

Issue H-7: Executing funded standard proposals can be prevented by a proposal slate with duplicate proposals

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/119

Found by

berndartmueller

Summary

Anyone can propose a slate of standard proposals to be funded in a distribution period with the `StandardFunding.checkSlate` function. The proposal slate can contain duplicate proposal ids, which, if the slate is the top slate, can be used to prevent a standard proposal from being executed (funded).

Vulnerability Detail

A funded standard proposal is executed by calling the `StandardFunding.executeStandard` function. A proposal is considered successfully funded if its state returned by the `GrantFund.state` function is `IGovernor.ProposalState.Succeeded`. This is the case if `StandardFunding._standardFundingVoteSucceeded` returns `true`.

`StandardFunding._standardFundingVoteSucceeded` checks if the given proposal id is included in the currently funded proposal slate.

However, as mentioned in the beginning, the proposal slate can contain duplicate proposal ids. A slate can therefore be maximized (in regard to the allocated budget) with the same proposal id. Worst case, this "malicious" slate can not be replaced with a correct slate, as the allocated budget of a correct slate can not exceed the allocated budget of the malicious slate.

In this case, the `StandardFunding.executeStandard` function will not execute the proposal, which is not included in the "malicious" proposal slate.

Impact

Standard proposals can be prevented from being funded in a distribution period.

Code Snippet

ecosystem-coordination/src/grants/base/StandardFunding.sol#L198-L219

```
198: for (uint i = 0; i < proposalIds_.length; ) {
199:     // check if Proposal is in the topTenProposals list
```

```
    if (_findProposalIndex(proposalIds_[i], topTenProposals[distributionId_]) == -1) return false;

    Proposal memory proposal = standardFundingProposals[proposalIds_[i]];

    // account for qvBudgetAllocated possibly being negative
    if (proposal.qvBudgetAllocated < 0) return false;

    // update counters
    sum += uint256(proposal.qvBudgetAllocated);
    totalTokensRequested += proposal.tokensRequested;

    // check if slate of proposals exceeded budget constraint ( 90% of GBC )
    if (totalTokensRequested > (gbc * 9 / 10)) {
        return false;
    }

    unchecked {
        ++i;
    }
}
```

Tool used

Manual Review

Recommendation

Consider checking for duplicate proposal ids in the `checkSlate` function.

Issue H-8: Adversary can grief kicker by frontrunning kick-Auction call with a large amount of loan

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/111

Found by

koxuan

Summary

Average debt size of the pool is used to calculated MOMP (Most optimistic matching price), which is used to derive NP (neutral price). Higher average debt size will result in lower MOMP and hence lower NP which will make it harder for kicker to earn a reward and more likely that the kicker is penalized. An adversary can manipulate the average debt size of the pool by frontrunning kicker's `kickAuction` call with a large amount of loan.

Vulnerability Detail

NP (neutral price) is a price that will be used to decide whether to reward a kicker with a bonus or punish the kicker with a penalty. In the event the auction ends with a price higher than NP, kicker will be given a penalty and if the auction ends with a price lower than NP, kicker will be rewarded with a bonus.

NP is derived from MOMP (Most optimistic matching price). BI refers to borrower inflator. Quoted from the whitepaper page 17, When a loan is initiated (the first debt or additional debt is drawn, or collateral is removed from the loan), the neutral price is set to the current MOMP times the ratio of the loan's threshold price to the LUP, plus one year's interest. As time passes, the neutral price increases at the same rate as interest. This can be expressed as the following formula for the neutral price as a function of time , where is the time the loan is initiated.

Therefore the lower the MOMP, the lower the NP. Lower NP will mean that kicker will be rewarded less and punished more compared to a higher NP. Quoted from the white paper, The MOMP, or "most optimistic matching price," is the price at which a loan of average size would match with the most favorable lenders on the book. Technically, it is the highest price for which the amount of deposit above it exceeds the average loan debt of the pool. In `_kick` function, MOMP is calculated as this. Notice how total pool debt is divided by number of loans to find the average loan debt size.

```
uint256 momp = _priceAt(
    Deposits.findIndexOfSum(
        deposits_,
        Maths.wdiv(poolState_.debt, noOfLoans * 1e18)
```

```
    )
);
```

An adversary can frontrun `kickAuction` by taking a huge loan, causing the price for which the amount of deposit above the undercollaterized loan bucket to have a lower probability of surpassing the average loan debt. The adversary can use the deposits for the buckets above and the total pool debt to figure out how much loan is necessary to grief the kicker significantly by lowering the MOMP and NP.

Impact

Kickers can be grieved which can disincentivize user from kicking loans that deserve to be liquidated, causing the protocol to not work as desired as undercollaterized loans will not be liquidated.

Code Snippet

Auctions.sol#L796-L801

Tool used

Manual Review

Recommendation

Recommend taking the snapshot average loan size of the pool to prevent frontrunning attacks.

Issue H-9: ERC721Pool's mergeOrRemoveCollateral allows to remove collateral while auction is clearable

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/105

Found by

hyh

Summary

User facing mergeOrRemoveCollateral() can effectively remove collateral, but lacks _revertIfAuctionClearable() check, i.e. allows to remove it while auction wasn't cleared.

Vulnerability Detail

mergeOrRemoveCollateral() can remove collateral funds from the pool with _transferFromPoolToAddress() when the amount requested has been merged. As settling the auction can alter the collateral in some buckets its removal is generally restricted in the protocol when auction wasn't yet cleared.

Impact

Collateral can be removed while auction result can change the allocation of the collateral in the buckets, i.e. can alter collateral amount per LP shares. This way mergeOrRemoveCollateral() result will not correspond to the current state of the pool and will lead to either a lender who initiated the call benefiting at the expense of other bucket LPs or vice versa, caller will have less collateral for the LP shares spent as some will be added to the bucket as a result of auction settlement.

Either way it is a gain for some LP at the expense of the others and a distribution based on a stale pool state (i.e. without auction result). As mergeOrRemoveCollateral() can be called by a lender at will an attacker will use it exactly when it is beneficial, at the expense of other participants.

Due to that setting the severity to be high as auctions are regular and so there is no low probability prerequisites.

Code Snippet

mergeOrRemoveCollateral() allows for removing a collateral when the auction is clearable:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L280-L322

```
function mergeOrRemoveCollateral(
    uint256[] calldata removalIndexes_,
    uint256 noOfNFTsToRemove_,
    uint256 toIndex_
) external override nonReentrant returns (uint256 collateralMerged_, uint256
↪   bucketLPs_) {
    PoolState memory poolState = _accruePoolInterest();
    uint256 collateralAmount = Maths.wad(noOfNFTsToRemove_);

    (
        collateralMerged_,
        bucketLPs_
    ) = LenderActions.mergeOrRemoveCollateral(
        buckets,
        deposits,
        removalIndexes_,
        collateralAmount,
        toIndex_
    );

    emit MergeOrRemoveCollateralNFT(msg.sender, collateralMerged_, bucketLPs_);

    // update pool interest rate state
    _updateInterestState(poolState, _lup(poolState.debt));

    if (collateralMerged_ == collateralAmount) {
        // Total collateral in buckets meets the requested removal amount,
↪   noOfNFTsToRemove_
        _transferFromPoolToAddress(msg.sender, bucketTokenIds,
↪   noOfNFTsToRemove_);
    }

}

/**
 *  @inheritdoc IPoolLenderActions
 *  @dev write state:
 *          - update bucketTokenIds arrays
 *  @dev emit events:
 *          - RemoveCollateral
 */
function removeCollateral(
    uint256 noOfNFTsToRemove_,
    uint256 index_
) external override nonReentrant returns (uint256 collateralAmount_, uint256
↪   lpAmount_) {
```

```
    _revertIfAuctionClearable(auctions, loans);
```

_revertIfAuctionClearable() checks whether pool state is ready to be altered by a result of the current auction:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/helpers/RevertsHelper.sol#L47-L59

```
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();

        Borrower storage borrower = loans_.borrowers[head];
        if (borrower.t0Debt != 0 && borrower.collateral == 0) revert
↪   AuctionNotCleared();
    }
}
```

Tool used

Manual Review

Recommendation

Consider adding the check:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L280-L286

```
    function mergeOrRemoveCollateral(
        uint256[] calldata removalIndexes_,
        uint256 noOfNFTsToRemove_,
        uint256 toIndex_
    ) external override nonReentrant returns (uint256 collateralMerged_, uint256
↪   bucketLPs_) {
+       _revertIfAuctionClearable(auctions, loans);
        PoolState memory poolState = _accruePoolInterest();
        uint256 collateralAmount = Maths.wad(noOfNFTsToRemove_);
```

Issue H-10: Remaining collateral used by ERC721Pool is missed in Auctions take and bucketTake return structures

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/103

Found by

hyh

Summary

ERC721Pool's take() and bucketTake() use remaining collateral variable to adjust borrower's collateral ids array after their debt is settled in an auction. In both cases this variable isn't initialized in Auctions's take() and bucketTake() and all collateral of the borrower ends up being frozen in the pool.

Vulnerability Detail

Being run with uninitialized zero `result.remainingCollateral`, _rebalanceTokens() will remove all the collateral ids from the borrower and place them into `bucketTokenIds`, i.e. move all borrowers collateral to the LP's cumulative collateral account. Those funds will be permanently frozen as other accounting parts will not have them recorded in any bucket, so no LP be able to withdraw extra funds.

Impact

Borrower's collateral funds will be permanently frozen in full whenever take() and bucketTake() result in auction settlement.

This takes place as after those ids was removed from `borrowerTokenIds` any operation of the borrower that should result in collateral being returned to them will be reverted instead.

Code Snippet

While being used in `_rebalanceTokens(borrowerAddress_, result.remainingCollateral)` call, the `result.remainingCollateral` isn't initialized and is always zero:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L405-L460

```
function take(
    address         borrowerAddress_,
    uint256         collateral_,
    address         callee_,
```


SHERLOCK

```
    bytes calldata data_
) external override nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

    TakeResult memory result = Auctions.take(
        auctions,
        buckets,
        deposits,
        loans,
        poolState,
        borrowerAddress_,
        Maths.wad(collateral_),
        1
    );

    // update pool balances state
    uint256 t0PoolDebt      = poolBalances.t0Debt;
    uint256 t0DebtInAuction = poolBalances.t0DebtInAuction;

    if (result.t0DebtPenalty != 0) {
        t0PoolDebt      += result.t0DebtPenalty;
        t0DebtInAuction += result.t0DebtPenalty;
    }

    t0PoolDebt      -= result.t0RepayAmount;
    t0DebtInAuction -= result.t0DebtInAuctionChange;

    poolBalances.t0Debt            =  t0PoolDebt;
    poolBalances.t0DebtInAuction   =  t0DebtInAuction;
    poolBalances.pledgedCollateral -= result.collateralAmount;

    // update pool interest rate state
    poolState.debt       =  result.poolDebt;
    poolState.collateral -= result.collateralAmount;
    _updateInterestState(poolState, result.newLup);

    // transfer rounded collateral from pool to taker
    uint256[] memory tokensTaken = _transferFromPoolToAddress(
        callee_,
        borrowerTokenIds[borrowerAddress_],
        result.collateralAmount / 1e18
    );

    if (data_.length != 0) {
        IERC721Taker(callee_).atomicSwapCallback(
            tokensTaken,
            result.quoteTokenAmount / _getArgUint256(QUOTE_SCALE),
            data_
```

```
        );
    }

    if (result.settledAuction) _rebalanceTokens(borrowerAddress_,
↪   result.remainingCollateral);
```

The reason is take doesn't fill `result.remainingCollateral = borrower.collateral` in the end of function:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L538-L594

```
function take(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState storage loans_,
    PoolState memory poolState_,
    address borrowerAddress_,
    uint256 collateral_,
    uint256 collateralScale_
) external returns (TakeResult memory result_) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    // revert if borrower's collateral is 0 or if maxCollateral to be taken is 0
    if (borrower.collateral == 0 || collateral_ == 0) revert
↪   InsufficientCollateral();

    (
        result_.collateralAmount,
        result_.quoteTokenAmount,
        result_.t0RepayAmount,
        borrower.t0Debt,
        result_.t0DebtPenalty,
        result_.excessQuoteToken
    ) = _take(
        auctions_,
        TakeParams({
            borrower:        borrowerAddress_,
            collateral:      borrower.collateral,
            t0Debt:          borrower.t0Debt,
            takeCollateral:  collateral_,
            inflator:        poolState_.inflator,
            poolType:        poolState_.poolType,
            collateralScale: collateralScale_
        })
    );
```

```
    borrower.collateral -= result_.collateralAmount;

    if (result_.t0DebtPenalty != 0) {
        poolState_.debt += Maths.wmul(result_.t0DebtPenalty,
 ↪  poolState_.inflator);
    }

    (
        result_.poolDebt,
        result_.newLup,
        result_.t0DebtInAuctionChange,
        result_.settledAuction
    ) = _takeLoan(
        auctions_,
        buckets_,
        deposits_,
        loans_,
        poolState_,
        borrower,
        borrowerAddress_,
        result_.t0RepayAmount
    );
}
```

Same for bucketTake(), it's used, but not filled, so `result.remainingCollateral` is always uninitialized zero:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L476-L518

```
function bucketTake(
    address borrowerAddress_,
    bool    depositTake_,
    uint256 index_
) external override nonReentrant {

    PoolState memory poolState = _accruePoolInterest();

    BucketTakeResult memory result = Auctions.bucketTake(
        auctions,
        buckets,
        deposits,
        loans,
        poolState,
        borrowerAddress_,
        depositTake_,
        index_,
        1
```

```
    );

    // update pool balances state
    uint256 t0PoolDebt      = poolBalances.t0Debt;
    uint256 t0DebtInAuction = poolBalances.t0DebtInAuction;

    if (result.t0DebtPenalty != 0) {
        t0PoolDebt      += result.t0DebtPenalty;
        t0DebtInAuction += result.t0DebtPenalty;
    }

    t0PoolDebt      -= result.t0RepayAmount;
    t0DebtInAuction -= result.t0DebtInAuctionChange;

    poolBalances.t0Debt            =  t0PoolDebt;
    poolBalances.t0DebtInAuction   =  t0DebtInAuction;
    poolBalances.pledgedCollateral -= result.collateralAmount;

    // update pool interest rate state
    poolState.debt       = result.poolDebt;
    poolState.collateral -= result.collateralAmount;
    _updateInterestState(poolState, result.newLup);

    if (result.settledAuction) _rebalanceTokens(borrowerAddress_,
 ↪  result.remainingCollateral);
}
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L463-L528

```
/**
 *  @notice Performs bucket take collateral on an auction, rewards taker and
 ↪  kicker (if case) and updates loan info (settles auction if case).
 *  @dev    reverts on:
 *              - insufficient collateral InsufficientCollateral()
 *  @param  borrowerAddress_ Borrower address to take.
 *  @param  depositTake_     If true then the take will happen at an auction
 ↪  price equal with bucket price. Auction price is used otherwise.
 *  @param  index_           Index of a bucket, likely the HPB, in which
 ↪  collateral will be deposited.
 *  @return result_          BucketTakeResult struct containing details of take.
*/
function bucketTake(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState storage loans_,
```

```
    PoolState memory poolState_,
    address borrowerAddress_,
    bool    depositTake_,
    uint256 index_,
    uint256 collateralScale_
) external returns (BucketTakeResult memory result_) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    if (borrower.collateral == 0) revert InsufficientCollateral(); // revert if
↪   borrower's collateral is 0

    (
        result_.collateralAmount,
        result_.t0RepayAmount,
        borrower.t0Debt,
        result_.t0DebtPenalty
    ) = _takeBucket(
        auctions_,
        buckets_,
        deposits_,
        BucketTakeParams({
            borrower:        borrowerAddress_,
            collateral:      borrower.collateral,
            t0Debt:          borrower.t0Debt,
            inflator:        poolState_.inflator,
            depositTake:     depositTake_,
            index:           index_,
            collateralScale: collateralScale_
        })
    );

    borrower.collateral -= result_.collateralAmount;

    if (result_.t0DebtPenalty != 0) {
        poolState_.debt += Maths.wmul(result_.t0DebtPenalty,
↪   poolState_.inflator);
    }

    (
        result_.poolDebt,
        result_.newLup,
        result_.t0DebtInAuctionChange,
        result_.settledAuction
    ) = _takeLoan(
        auctions_,
        buckets_,
        deposits_,
        loans_,
```

```
        poolState_,
        borrower,
        borrowerAddress_,
        result_.t0RepayAmount
    );
}
```

As a result _rebalanceTokens() will leave only `borrowerCollateral_ = result.remainingCollateral = 0` in the borrower's collateral ids account:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L533-L550

```
function _rebalanceTokens(
    address borrowerAddress_,
    uint256 borrowerCollateral_
) internal {
    // rebalance borrower's collateral, transfer difference to floor collateral
    ↪ from borrower to pool claimable array
    uint256[] storage borrowerTokens = borrowerTokenIds[borrowerAddress_];

    uint256 noOfTokensPledged    = borrowerTokens.length;
    uint256 noOfTokensToTransfer = borrowerCollateral_ != 0 ? noOfTokensPledged
    ↪ - borrowerCollateral_ / 1e18 : noOfTokensPledged;

    for (uint256 i = 0; i < noOfTokensToTransfer;) {
        uint256 tokenId = borrowerTokens[--noOfTokensPledged]; // start with
    ↪ moving the last token pledged by borrower
        borrowerTokens.pop();                                  // remove token
    ↪ id from borrower
        bucketTokenIds.push(tokenId);                          // add token id
    ↪ to pool claimable tokens

        unchecked { ++i; }
    }
}
```

Tool used

Manual Review

Recommendation

Consider filling the variable with the resulting collateral of the borrower by placing `result.remainingCollateral = borrower.collateral` in the very end of Auctions's take() and bucketTake().

Issue H-11: ERC721Pool's take will proceed with truncated collateral amount and full debt when borrower's collateral is fractional

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/68

Found by

hyh

Summary

Caller of take() can end up paying the debt corresponding to the fractional ERC721 collateral of a borrower, but receiving only truncated part of the this collateral in return (paying the debt for `1.9`, receiving `1.0`), with the borrower keeping the remainder.

Vulnerability Detail

Fractional part of ERC721 collateral is gifted to the borrower in Auctions's _take() (L889-898) when `params_.collateral` doesn't allow an increase. Say when `vars.collateralAmount = params_.collateral = 1.9e18`, while taker specified collateral is `2`, it will proceed with paying the debt corresponding to `1.9e18`, which was calculated before in _calculateTakeFlowsAndBondChange(), but will pay the caller only `1e18` of collateral, leaving `0.9e18` with the borrower at caller's expense.

It happens only when `params_.collateral = borrower.collateral` isn't whole 18dp integer, the state that can periodically occur after ERC721Pool's bucketTake(), which applies _calculateTakeFlowsAndBondChange() result to the borrower's balance without rounding, so a partial bucketTake() will leave it as a 18dp fraction.

Impact

Caller's funds will be lost as they pay borrower's debt according to the untruncated `params_.collateral` value, but receive only truncated amount of collateral.

As both take() and bucketTake() are routine operations and there are no low probability prerequisites, and given the loss of funds for the taker, setting the severity to be high.

Code Snippet

Debt is calculated off `min(params_.collateral, params_.takeCollateral)`, but if `params_.collateral` is a fraction, say `1.9e18`, the `0.9e18` of collateral is gifted back to the borrower:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L854-L909

```
function _take(
    AuctionsState storage auctions_,
    TakeParams memory params_
) internal returns (uint256, uint256, uint256, uint256, uint256, uint256) {
    ...

    vars = _calculateTakeFlowsAndBondChange(
        Maths.min(params_.collateral, params_.takeCollateral),
        params_.inflator,
        params_.collateralScale,
        vars
    );

    ...

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

        if (collateralTaken != vars.collateralAmount && params_.collateral >=
↪   collateralTaken + 1e18) { // collateral taken not a round number
            collateralTaken += 1e18; // round up collateral to take
            // taker should send additional quote tokens to cover difference
↪   between collateral needed to be taken and rounded collateral, at auction
↪   price
            // borrower will get quote tokens for the difference between rounded
↪   collateral and collateral taken to cover debt
            vars.excessQuoteToken = Maths.wmul(collateralTaken -
↪   vars.collateralAmount, vars.auctionPrice);
        }

        vars.collateralAmount = collateralTaken;
    }

    return ...;
}
```

`params_.takeCollateral` is caller specified collateral value, while `params_.collateral` is `borrower.collateral`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L538-L571

```
function take(
```

```
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState storage loans_,
    PoolState memory poolState_,
    address borrowerAddress_,
    uint256 collateral_,
    uint256 collateralScale_
) external returns (TakeResult memory result_) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    // revert if borrower's collateral is 0 or if maxCollateral to be taken is 0
    if (borrower.collateral == 0 || collateral_ == 0) revert
↪   InsufficientCollateral();

    (
        result_.collateralAmount,
        result_.quoteTokenAmount,
        result_.t0RepayAmount,
        borrower.t0Debt,
        result_.t0DebtPenalty,
        result_.excessQuoteToken
    ) = _take(
        auctions_,
        TakeParams({
            borrower:        borrowerAddress_,
            collateral:      borrower.collateral,
            t0Debt:          borrower.t0Debt,
            takeCollateral:  collateral_,
            inflator:        poolState_.inflator,
            poolType:        poolState_.poolType,
            collateralScale: collateralScale_
        })
    );
```

Caller specified `params_.takeCollateral = collateral_` of ERC721Pool's take() is always whole 1e18 integer via `Maths.wad(collateral_)`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L405-L422

```
function take(
    address        borrowerAddress_,
    uint256        collateral_,
    address        callee_,
    bytes calldata data_
) external override nonReentrant {
    PoolState memory poolState = _accruePoolInterest();
```

```
    TakeResult memory result = Auctions.take(
        auctions,
        buckets,
        deposits,
        loans,
        poolState,
        borrowerAddress_,
        Maths.wad(collateral_),
        1
    );
```

But `params_.collateral = borrower.collateral` can be fractional as bucketTake() do not round collateral result in the ERC721 case and subtract this result from `borrower.collateral`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L472-L507

```
function bucketTake(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState storage loans_,
    PoolState memory poolState_,
    address borrowerAddress_,
    bool    depositTake_,
    uint256 index_,
    uint256 collateralScale_
) external returns (BucketTakeResult memory result_) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    if (borrower.collateral == 0) revert InsufficientCollateral(); // revert if
↪   borrower's collateral is 0

    (
        result_.collateralAmount,
        result_.t0RepayAmount,
        borrower.t0Debt,
        result_.t0DebtPenalty
    ) = _takeBucket(
        auctions_,
        buckets_,
        deposits_,
        BucketTakeParams({
            borrower:       borrowerAddress_,
            collateral:     borrower.collateral,
            t0Debt:         borrower.t0Debt,
```

```
            inflator:         poolState_.inflator,
            depositTake:      depositTake_,
            index:            index_,
            collateralScale: collateralScale_
        })
    );

    borrower.collateral -= result_.collateralAmount;
```

Tool used

Manual Review

Recommendation

One way is to revert such take attempts:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L887-L899

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

-           if (collateralTaken != vars.collateralAmount && params_.collateral
↪   >= collateralTaken + 1e18) { // collateral taken not a round number
+           if (collateralTaken != vars.collateralAmount) { // collateral taken
↪   not a round number
+               if (params_.collateral >= collateralTaken + 1e18) {
                    collateralTaken += 1e18; // round up collateral to take
                    // taker should send additional quote tokens to cover
↪   difference between collateral needed to be taken and rounded collateral, at
↪   auction price
                    // borrower will get quote tokens for the difference between
↪   rounded collateral and collateral taken to cover debt
                    vars.excessQuoteToken = Maths.wmul(collateralTaken -
↪   vars.collateralAmount, vars.auctionPrice);
+                   vars.collateralAmount = collateralTaken;
+               } else {
+                   revert collateralRoundingIsNeededButNotPossible();
+               }
            }

-           vars.collateralAmount = collateralTaken;
        }
```

A drawback is that, while the taker can repeat the call with `1` collateral and succeed with it, the `0.9` part will be untakeable. It looks to be a natural limitation of using ERC721 collaterals. From this point some borrower collateral pooling mechanics can be accessed similar to the existing bucket's fractional collateral pooling logic.

Early revert when borrower's collateral is less than `1` can be advised if the pool is ERC721:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L538-L552

```
function take(
    ...
) external returns (TakeResult memory result_) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    // revert if borrower's collateral is 0 or if maxCollateral to be taken is 0
    if (borrower.collateral == 0 || collateral_ == 0) revert
↪   InsufficientCollateral();
```

Issue H-12: The deposit / withdraw / trade transaction lack of expiration timestamp check and slippage control

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/39

Found by

ctf_sec

Summary

The deposit / withdraw / trade transaction lack of expiration timestamp and slippage control

Vulnerability Detail

Let us look into the heavily forked Uniswap V2 contract addLiquidity function implementation

https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router02.sol#L61

```
// **** ADD LIQUIDITY ****
function _addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin
) internal virtual returns (uint amountA, uint amountB) {
    // create the pair if it doesn't exist yet
    if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
        IUniswapV2Factory(factory).createPair(tokenA, tokenB);
    }
    (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
↪   tokenA, tokenB);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
↪   reserveB);
        if (amountBOptimal <= amountBDesired) {
            require(amountBOptimal >= amountBMin, 'UniswapV2Router:
↪   INSUFFICIENT_B_AMOUNT');
            (amountA, amountB) = (amountADesired, amountBOptimal);
```

```
        } else {
            uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
 ↪  reserveB, reserveA);
            assert(amountAOptimal <= amountADesired);
            require(amountAOptimal >= amountAMin, 'UniswapV2Router:
 ↪  INSUFFICIENT_A_AMOUNT');
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}

function addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint amountA, uint
 ↪  amountB, uint liquidity) {
    (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
 ↪  amountBDesired, amountAMin, amountBMin);
    address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
    TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
    TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IUniswapV2Pair(pair).mint(to);
}
```

the implementation has two point that worth noting,

the first point is the deadline check

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

The transaction can be pending in mempool for a long and the trading activity is very time senstive. Without deadline check, the trade transaction can be executed in a long time after the user submit the transaction, at that time, the trade can be done in a sub-optimal price, which harms user's position.

The deadline check ensure that the transaction can be executed on time and the expired transaction revert.

the second point is the slippage control:

```
require(amountAOptimal >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
```

and

```
require(amountBOptimal >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');
```

the slippage control the user can receive the least optimal amount of the token they want to trade.

In the current implementation, neither the deadline check nor the slippage control is in place when user deposit / withdraw / trade.

Impact

According to the whitepaper:

> Deposits in the highest priced buckets offer the highest valuations on collateral, and hence offer the most liquidity to borrowers. They are also the first buckets that could be used to purchase collateral if a loan were to be liquidated (see 7.0 LIQUIDATIONS). We can think of a bucket's deposit as being utilized if the sum of all deposits in buckets priced higher than it is less than the total debt of all borrowers in the pool. The lowest price among utilized buckets or "lowest utilized price" is called the LUP. If we were to pair off lenders with borrowers, matching the highest priced lenders' deposits with the borrowers' debts in equal quantities, the LUP would be the price of the marginal (lowest priced and therefore least aggressive) lender thus matched (usually, there would be a surplus of lenders that were not matched, corresponding to less than 100% utilization of the pool).
>
> The LUP plays a critical role in Ajna: a borrower who is undercollateralized with respect to the LUP (i.e. with respect to the marginal utilized lender) is eligible for liquidation. Conversely, a lender cannot withdraw deposit if doing so would move the LUP down so far as to make some active loans eligible for liquidation. In order to withdraw quote token in this situation, the lender must first kick the loans in question.

Because the deadline check is missing,

After a lender submit a transaction and want to add the token into Highest price busket to make sure the quote token can be borrowed out and generate yield.

However, the transaction is pending in the mempool for a very long time.

Borrower create more debt and other lender's add and withdraw quote token before the lender's transaction is executed.

After a long time later, the lender's transaction is executed.

The lender find out that the highest priced bucket moved and the lender cannot withdraw his token because doing would move the LUP down eligible for liquidiation.

According to the whitepaper:

> 6.1 Trading collateral for quote token
>
> David owns 1 ETH, and would like to sell it for 1100 DAI. He puts the 1 ETH into the 1100 bucket as claimable collateral (alongside Carol's 20000 deposit), minting 1100 in LPB in return. He can then redeem that 1100 LPB for quote token, withdrawing 1100 DAI. Note: after David's withdrawal, the LUP remains at 1100. If the book were different such that his withdrawal would move the LUP below Bob's threshold price of 901.73, he would not be able to withdraw all of the DAI.

The case above is ideal, however, because the deadline check is missing, and there is no slippage control, the transactoin can be pending for a long time and by the time the trade transaction is lended, the withdraw amount can be less than 1100 DAI.

Another example for lack of slippage, for example, the function below is called:

```
/// @inheritdoc IPoolLenderActions
function removeQuoteToken(
    uint256 maxAmount_,
    uint256 index_
) external override nonReentrant returns (uint256 removedAmount_, uint256
↪   redeemedLPs_) {
    _revertIfAuctionClearable(auctions, loans);

    PoolState memory poolState = _accruePoolInterest();

    _revertIfAuctionDebtLocked(deposits, poolBalances, index_,
↪   poolState.inflator);

    uint256 newLup;
    (
        removedAmount_,
        redeemedLPs_,
        newLup
    ) = LenderActions.removeQuoteToken(
        buckets,
        deposits,
        poolState,
        RemoveQuoteParams({
            maxAmount:      maxAmount_,
```

```
            index:          index_,
            thresholdPrice: Loans.getMax(loans).thresholdPrice
        })
    );

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

    // move quote token amount from pool to lender
    _transferQuoteToken(msg.sender, removedAmount_);
}
```

without specificing the minReceived amount, the removedAmount can be very small comparing to the maxAmount user speicifced.

Code Snippet

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L130-L158

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L202

Tool used

Manual Review

Recommendation

We recommend the protocol add deadline check and add slippage control.

Issue M-1: Buypunk function of Cryptopunks in ERC721Pool is used incorrectly

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/163

Found by

Chinmay

Summary

The buyPunk function here seems to be for transferring NFT from sender to pool, but the original contract has a payable function that uses msg.value checks

Vulnerability Detail

This seems to be a weird implementation for transferring the NFT. Furthermore, the function is payable but the interface by AJNA doesn't mark it as payable.

This function checks for the msg.value in the original Cryptopunks contract. Calling it from the ERC721Pool will always revert because the msg.value is not being sent with the call at L#577. Thus, a cryptopunk NFT will never be able to be used as the collateral in this NFT pool.

Impact

Code Snippet

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L577

Tool used

Manual Review

Recommendation

Update the interface with the payable keyword and send msg.value along with the buyPunk call so that it passes checks at the target contract

Discussion

grandizzy

we're not going to support non standard NFT anymore, just wrapped versions

Issue M-2: ERC721Pool taker callback misreports quote funds whenever there was collateral amount rounding

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/162

Found by

hyh

Summary

atomicSwapCallback() now reports `tokensTaken` higher than one corresponding to `quoteTokenAmount` whenever the rounding took place as the `excessQuoteToken` is omitted.

Vulnerability Detail

ERC721Pool's take() `atomicSwapCallback` needs to have full quote amount, `(result.quoteTokenAmount + result.excessQuoteToken)`, when there is a quote token part added to have the whole integer amount of the collateral asset.

Now the callback quote value, `quoteTokenAmount`, is inconsistent with the collateral amount `tokensTaken` when there was rounding.

Impact

The impact depends on the logic on `callee_` side, but asset amounts are usually used in the downstream asset management logic, so misreporting such amounts can lead to the asset losses on the taker's side.

As that's the precondition, setting the severity to be medium.

Code Snippet

ERC721Pool's take() reports `result.quoteTokenAmount` and `tokensTaken` as take result to the `callee_`, ignoring collateral rounding part:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L452-L463

```
if (data_.length != 0) {
    IERC721Taker(callee_).atomicSwapCallback(
        tokensTaken,
        result.quoteTokenAmount / _getArgUint256(QUOTE_SCALE),
        data_
    );
```

```
}

if (result.settledAuction) _rebalanceTokens(borrowerAddress_,
↪  result.remainingCollateral);

// transfer from taker to pool the amount of quote tokens needed to cover
↪  collateral auctioned (including excess for rounded collateral)
_transferQuoteTokenFrom(callee_, result.quoteTokenAmount +
↪  result.excessQuoteToken);
```

Tool used

Manual Review

Recommendation

Consider adding the `excessQuoteToken` to the reported value:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L452-L463

```
        if (data_.length != 0) {
            IERC721Taker(callee_).atomicSwapCallback(
                tokensTaken,
-               result.quoteTokenAmount / _getArgUint256(QUOTE_SCALE),
+               (result.quoteTokenAmount + result.excessQuoteToken) /
↪  _getArgUint256(QUOTE_SCALE),
                data_
            );
        }

        if (result.settledAuction) _rebalanceTokens(borrowerAddress_,
↪  result.remainingCollateral);

        // transfer from taker to pool the amount of quote tokens needed to
↪  cover collateral auctioned (including excess for rounded collateral)
        _transferQuoteTokenFrom(callee_, result.quoteTokenAmount +
↪  result.excessQuoteToken);
```

Issue M-3: Anyone can transfer approved LP tokens

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/156

Found by

Jeiwan

Summary

Anyone can call the `Pool.transferLPs` function and transfer previously approved LP tokens to the approved address.

Vulnerability Detail

The Pool.transferLPs function allows to transfer LP tokens from one address to another. Even though it requires approving a transfer, actual transferring is left at the discretion of the approved address: approving allows the approved address to transfer LP tokens when appropriate. However, since the `Pool.transferLPs` function can be called by any address, the owner of the tokens may be impacted.

Impact

Lender's LP tokens may be transferred to an approve address at an inappropriate time, impacting the position management strategy of the lender.

Code Snippet

Pool.sol#L238

Tool used

Manual Review

Recommendation

Consider allowing calling the `Pool.transferLPs` function only to the `owner` or `newOwner_`.

Discussion

grandizzy

removing will fix label, will address after Sherlock contest

Issue M-4: Incorrect MOMP calculation in neutral price calculation

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/148

Found by

Jeiwan

Summary

When calculating MOMP to find the neutral price of a borrower, borrower's accrued debt is divided by the total number of loans in the pool, but it's total pool's debt that should be divided. The mistake will result in lower neutral prices and more lost bonds to kickers.

Vulnerability Detail

As per the whitepaper:

> MOMP: is the price at which the amount of deposit above it is equal to the average loan size of the pool. MOMP is short for "Most Optimistic Matching Price", as it's the price at which a loan of average size would match with the most favorable lenders on the book.

I.e. MOMP is calculated on the total number of loans of a pool (so that the average loan size could be found).

MOMP calculation is implemented correctly when kicking a debt, however it's implementation in the Loans.update function is not correct:

```
uint256 loansInPool = loans_.loans.length - 1 + auctions_.noOfAuctions;
uint256 curMomp     = _priceAt(Deposits.findIndexOfSum(deposits_,
↪  Maths.wdiv(borrowerAccruedDebt_, loansInPool * 1e18)));
```

Here, only borrower's debt (`borrowerAccruedDebt_`) is divided, not the entire debt of the pool.

Impact

The miscalculation affects only borrower's neutral price calculation. Since MOMP is calculated on a smaller debt (borrower's debt will almost always be smaller than total pool's debt), the value of MOMP will be smaller than expected, and the neutral price will also be smaller (from the whitepaper: "The NP of a loan is the interest-adjusted MOMP..."). This will cause kickers to lose their bonds more often than expected, as per the whitepaper:

> If the liquidation auction yields a value that is over the “Neutral Price,” NP, the kicker forfeits a portion or all of their bond.

Code Snippet

Loans.sol#L113-L114

Tool used

Manual Review

Recommendation

Consider using total pool's debt in the MOMP calculation in `Loans.update`.

Issue M-5: Extraordinary proposals can receive more tokens than eligible

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/124

Found by

berndartmueller

Summary

Two extraordinary proposals proposed at similar times (while no other extraordinary proposals are executed) can request the same amount of tokens (`proposal.tokensRequested`). But if the second proposal is proposed **after** the first proposal is successfully funded and executed, the second proposal could not request the same amount of tokens. It would have to be lower due to `ExtraordinaryFunding.sol#L86`.

Vulnerability Detail

Proposing an extraordinary funding proposal with the `ExtraordinaryFunding.proposeExtraordinary` function verifies that the requested token amount is within (less than) a certain limit in `ExtraordinaryFunding.sol#L86`. The limit is based on the treasury Ajna token balance and decreases with an increasing number of funded extraordinary proposals. A maximum number of **10** extraordinary proposals can be funded.

If multiple extraordinary proposals are proposed while the number of funded proposals is unchanged, the limit for the tokens requested is the same for those proposals. At a later time, if those proposals pass voting, the proposals are executed and receive the requested (stale) token amount.

Impact

The requested token amount of an extraordinary proposal is not checked when executing if it's within the same limits imposed by L86.

If another extraordinary proposal was successfully funded and executed in the meantime of the 1 month voting period for the proposal, the limit for the requested token amount is already lower than the `proposal.tokensRequested` amount.

This means that if multiple extraordinary proposals are proposed at a similar time and pass voting, the `tokensRequested` amount is stale and potentially too much.

Code Snippet

ecosystem-coordination/src/grants/base/ExtraordinaryFunding.sol#L86

```
067: function proposeExtraordinary(
068:     uint256 endBlock_,
069:     address[] memory targets_,
070:     uint256[] memory values_,
071:     bytes[] memory calldatas_,
072:     string memory description_) external returns (uint256 proposalId_) {
073:
074:     proposalId_ = hashProposal(targets_, values_, calldatas_, keccak256(bytes(description_)));
075:
076:     if (extraordinaryFundingProposals[proposalId_].proposalId != 0) revert ProposalAlreadyExists();
077:
078:     // check proposal length is within limits of 1 month maximum and it hasn't already been submitted
079:     if (block.number + MAX_EFM_PROPOSAL_LENGTH < endBlock_ || extraordinaryFundingProposals[proposalId_].proposalId != 0) {
080:         revert ExtraordinaryFundingProposalInvalid();
081:     }
082:
083:     uint256 totalTokensRequested = _validateCallDatas(targets_, values_, calldatas_);
084:
085:     // check tokens requested is within limits
086:     if (totalTokensRequested > getSliceOfTreasury(Maths.WAD - _getMinimumThresholdPercentage())) revert ExtraordinaryFundingProposalInvalid();
087:
088:     // store newly created proposal
089:     ExtraordinaryFundingProposal storage newProposal = extraordinaryFundingProposals[proposalId_];
090:     newProposal.proposalId      = proposalId_;
091:     newProposal.startBlock      = block.number;
092:     newProposal.endBlock        = endBlock_;
093:     newProposal.tokensRequested = totalTokensRequested;
094:
095:     emit ProposalCreated(
096:         proposalId_,
097:         msg.sender,
098:         targets_,
099:         values_,
100:         new string[](targets_.length),
101:         calldatas_,
102:         block.number,
```

```
        endBlock_,
        description_);
}

```

Tool used

Manual Review

Recommendation

Consider re-checking the `tokensRequested` amount when executing an extraordinary proposal and make sure it's within the same or similar limits as when proposing.

Issue M-6: Claiming rewards from a future not yet existing epoch prevents claiming rewards for those epochs later on

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/122

Found by

berndartmueller

Summary

If a user claims rewards for a future epoch, all epochs are marked as claimed up until that future epoch. This prevents the user from claiming rewards for those epochs later, leading to a loss of rewards.

Vulnerability Detail

Already claimed rewards are tracked in the `isEpochClaimed` mapping and checked in the `RewardsManager.claimRewards` function to prevent claiming rewards multiple times. However, the current implementation does not prevent a user from accidentally claiming rewards for a future epoch. This would iterate through all epochs up until the future epoch and mark them all as claimed. This prevents the user from claiming rewards for those epochs later on, leading to a loss of rewards.

Impact

If a user accidentally claims rewards for a future epoch, the rewards are lost and unclaimable.

Code Snippet

contracts/src/RewardsManager.sol#L112

```
106: function claimRewards(
107:     uint256 tokenId_,
108:     uint256 epochToClaim_
109: ) external override {
110:     if (msg.sender != stakes[tokenId_].owner) revert NotOwnerOfDeposit();
111:
112:     if (isEpochClaimed[tokenId_][epochToClaim_]) revert AlreadyClaimed();
113:
114:     _claimRewards(tokenId_, epochToClaim_);
115: }
```

contracts/src/RewardsManager.sol#L298

```
function _calculateAndClaimRewards(
    uint256 tokenId_,
    uint256 epochToClaim_
) internal returns (uint256 rewards_) {

    address ajnaPool       = stakes[tokenId_].ajnaPool;
    uint256 lastBurnEpoch = stakes[tokenId_].lastInteractionBurnEpoch;
    uint256 stakingEpoch  = stakes[tokenId_].stakingEpoch;

    uint256[] memory positionIndexes = positionManager.getPositionIndexes(tokenId_);

    // iterate through all burn periods to calculate and claim rewards
    for (uint256 epoch = lastBurnEpoch; epoch < epochToClaim_; ) {

        uint256 nextEpochRewards = _calculateNextEpochRewards(
            tokenId_,
            epoch,
            stakingEpoch,
            ajnaPool,
            positionIndexes
        );

        uint256 nextEpoch = epoch + 1;

        // update epoch token claim trackers
        rewardsClaimed[nextEpoch]           += nextEpochRewards;
        isEpochClaimed[tokenId_][nextEpoch] = true;

        rewards_ += nextEpochRewards;

        unchecked { ++epoch; }
    }
}
```

Tool used

Manual Review

Recommendation

Consider adding a check to the `RewardsManager.claimRewards` function to prevent claiming rewards for future epochs.

Discussion

grandizzy

Has #151 as dupe

Issue M-7: Calculating new rewards is susceptible to precision loss due to division before multiplication

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/121

Found by

berndartmueller

Summary

Rewards may be lost (0) due to division before multiplication precision issues.

Vulnerability Detail

The `RewardsManager._calculateNewRewards` function calculates the new rewards for a staker by first dividing `interestEarned_` by `totalInterestEarnedInPeriod` and then multiplying by `totalBurnedInPeriod`. If `interestEarned_` is small enough and `totalInterestEarnedInPeriod` is large enough, the division may result in a value of 0, resulting in the staker receiving 0 rewards.

Impact

Stakers may not receive rewards due to precision loss.

Code Snippet

contracts/src/RewardsManager.sol#L426-L428

```
408: function _calculateNewRewards(
409:     address ajnaPool_,
410:     uint256 interestEarned_,
411:     uint256 nextEpoch_,
412:     uint256 epoch_,
413:     uint256 rewardsClaimedInEpoch_
414: ) internal view returns (uint256 newRewards_) {
415:     (
416:         ,
417:         // total interest accumulated by the pool over the claim period
418:         uint256 totalBurnedInPeriod,
419:         // total tokens burned over the claim period
420:         uint256 totalInterestEarnedInPeriod
421:     ) = _getPoolAccumulators(ajnaPool_, nextEpoch_, epoch_);
422:
423:     // calculate rewards earned
```

```
    newRewards_ = Maths.wmul(
        REWARD_FACTOR,
        Maths.wmul(
            Maths.wdiv(interestEarned_, totalInterestEarnedInPeriod), totalBurnedInPeriod
        )
    );
```

Tool used

Manual Review

Recommendation

Consider calculating the new rewards by first multiplying `interestEarned_` by `totalBurnedInPeriod` and then dividing by `totalInterestEarnedInPeriod` to avoid precision loss.

Issue M-8: Claiming accumulated rewards while the contract is underfunded can lead to a loss of rewards

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/120

Found by

ctf_sec, oxcm, berndartmueller

Summary

The claimable rewards for an NFT staker are capped at the Ajna token balance at the time of claiming. This can lead to a loss of rewards if the `RewardsManager` contract is underfunded with Ajna tokens.

Vulnerability Detail

The `RewardsManager` contract keeps track of the rewards earned by an NFT staker. The accumulated rewards are claimed by calling the `RewardsManager.claimRewards` function. Internally, the `RewardsManager._claimRewards` function transfers the accumulated rewards to the staker.

However, the transferrable amount of Ajna token rewards are capped at the Ajna token balance at the time of claiming. If the accumulated rewards are higher than the Ajna token balance, the claimer will receive fewer rewards than expected. The remaining rewards cannot be claimed at a later time as the `RewardsManager` contract does not keep track of the rewards that were not transferred.

Impact

If an NFT staker claims the accumulated rewards in a bad situation (when the `RewardsManager` contract is underfunded with Ajna tokens), the staker will receive fewer rewards than expected and is unable to claim the rest of the rewards at a later time.

Code Snippet

contracts/src/RewardsManager.sol#L479

```
function _claimRewards(
    uint256 tokenId_,
    uint256 epochToClaim_
) internal {
    // [...]
```

```
    uint256 ajnaBalance = IERC20(ajnaToken).balanceOf(address(this));

    if (rewardsEarned > ajnaBalance) rewardsEarned = ajnaBalance;

    // transfer rewards to sender
    IERC20(ajnaToken).safeTransfer(msg.sender, rewardsEarned);
}
```

Tool used

Manual Review

Recommendation

Consider reverting if insufficient Ajna tokens are available as rewards.

Issue M-9: [M] Incorrect Validation in `Pool.sol#transferLPs` lead to a DOS attack

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/116

Found by

oxcm

Summary

The code in the transferLPs function has an incorrect validation check, where it requires `allowances_` to be strictly equal to `lenderLpBalance`, instead of just `allowances_` being greater than `transferAmount`.

Vulnerability Detail

In the `transferLPs()` function, `transferAmount` is being compared to `allowances_[owner_][newOwner_][index]` and `lenderLpBalance`. If the values are not strictly equal, the function will revert with a `NoAllowance` error.

Due to the requirement of `transferLPs()` that `allowances_` must equal `lenderLpBalance`, the user can only enter `lpsAmountToApprove_` as the current `lenderLpBalance` when using `approveLpOwnership()`.

This results in `transferLPs()` reverting with `NoAllowance` if `lenderLpBalance` undergoes any change, allowing attackers to design a DOS attack.

However, this validation is not necessary as it should only require `allowances_` to be greater than `transferAmount`.

Impact

An attacker could exploit this vulnerability by transferring a small amount of LP tokens to the owner before the transfer to the new owner is initiated. This would cause the `allowances_` value to be less than `lenderLpBalance`, causing the transfer to revert and the tokens to remain in the original owner's account.

Code Snippet

Relevant code snippet from transferLPs function:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L238-L250

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/LenderActions.sol#L512-L558

Tool used

Manual Review / ChatGPT

Recommendation

The validation check in the transferLPs function should be updated to allow for allowances_ to be greater than transferAmount, rather than requiring them to be strictly equal. The updated code would look like this:

```
if (transferAmount == 0 || allowances_[owner_][newOwner_][index] <
↪  transferAmount) revert NoAllowance();
```

and change `approveLpOwnership()` to:

```
function approveLpOwnership(
    address allowedNewOwner_,
    uint256 index_
) external nonReentrant {
    _lpTokenAllowances[msg.sender][allowedNewOwner_][index_] = type(uint256).max;
}
```

Discussion

grandizzy

removing will fix, will be addressed after sherlock contest

Issue M-10: Settled collateral of a borrower aren't available for lenders until borrower's debt is fully cleared

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/104

Found by

hyh

Summary

ERC721Pool's settle() lacks collateral ids array rebalance when the settlement isn't full, i.e. when `t0DebtRemaining > 0`.

Vulnerability Detail

Auctions's settlePoolDebt() returns current state of the borrower after the settlement. If any of borrower's collateral id were removed from them it needs to be accounted for, but when `t0DebtRemaining > 0` this doesn't happen, i.e. removed borrower's tokens aren't added to the buckets cumulative collateral ids array and so this collateral is still unavailable for lenders.

Impact

The settled collateral of the borrower will not be available for LP's withdrawal as the corresponding function will revert on an attempt to extract token ids from `bucketTokenIds` array.

The length of such freeze can vary up to be permanent, for example if there is no funds (reserves and deposits) to fully settle the borrower. This is principal fund loss scenario for the lenders, but given the prerequisite of the full default setting the severity to be medium.

Code Snippet

When a borrower has some unsettled debt, `t0DebtRemaining > 0`, all of their collateral remain locked with `borrowerTokenIds` as _rebalanceTokens() isn't called in this case:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L353-L385

```
function settle(
    address borrowerAddress_,
    uint256 maxDepth_
```

```
) external nonReentrant override {
    PoolState memory poolState = _accruePoolInterest();

    uint256 assets = Maths.wmul(poolBalances.t0Debt, poolState.inflator) +
↪   _getPoolQuoteTokenBalance();
    uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reserveAuction.unclaimed;

    SettleParams memory params = SettleParams(
        {
            borrower:    borrowerAddress_,
            reserves:    (assets > liabilities) ? (assets-liabilities) : 0,
            inflator:    poolState.inflator,
            bucketDepth: maxDepth_,
            poolType:    poolState.poolType
        }
    );
    (
        uint256 collateralRemaining,
        uint256 t0DebtRemaining,
        uint256 collateralSettled,
        uint256 t0DebtSettled
    ) = Auctions.settlePoolDebt(
        auctions,
        buckets,
        deposits,
        loans,
        params
    );

    // slither-disable-next-line incorrect-equality
    if (t0DebtRemaining == 0) _rebalanceTokens(params.borrower,
↪   collateralRemaining);
```

This can happen when settlement exits when `params_.bucketDepth == 0`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L277-L337

```
    // if there's still debt and no collateral
    if (borrower.t0Debt != 0 && borrower.collateral == 0) {
        // settle debt from reserves -- round reserves down however
        borrower.t0Debt -= Maths.min(borrower.t0Debt, (params_.reserves /
↪   params_.inflator) * 1e18);

        // if there's still debt after settling from reserves then start to
↪   forgive amount from next HPB
        // loop through remaining buckets if there's still debt to settle
```


SHERLOCK

```
        while (params_.bucketDepth != 0 && borrower.t0Debt != 0) {
            SettleLocalVars memory vars;

            (vars.index, , vars.scale) =
↪   Deposits.findIndexAndSumOfSum(deposits_, 1);
            vars.unscaledDeposit = Deposits.unscaledValueAt(deposits_,
↪   vars.index);
            vars.depositToRemove = Maths.wmul(vars.scale, vars.unscaledDeposit);
            vars.debt            = Maths.wmul(borrower.t0Debt, params_.inflator);

            // enough deposit in bucket to settle entire debt
            if (vars.depositToRemove >= vars.debt) {
                Deposits.unscaledRemove(deposits_, vars.index,
↪   Maths.wdiv(vars.debt, vars.scale));
                borrower.t0Debt  = 0;
↪                     // no remaining debt to settle

            // not enough deposit to settle entire debt, we settle only deposit
↪   amount
            } else {
                borrower.t0Debt -= Maths.wdiv(vars.depositToRemove,
↪   params_.inflator);             // subtract from remaining debt the
↪   corresponding t0 amount of deposit

                Deposits.unscaledRemove(deposits_, vars.index,
↪   vars.unscaledDeposit);               // Remove all deposit from bucket
                Bucket storage hpbBucket = buckets_[vars.index];

                if (hpbBucket.collateral == 0) {
↪                     // existing LPB and LP tokens for the bucket shall become
↪   unclaimable.
                    emit BucketBankruptcy(vars.index, hpbBucket.lps);
                    hpbBucket.lps            = 0;
                    hpbBucket.bankruptcyTime = block.timestamp;
                }
            }

            --params_.bucketDepth;
        }
    }

    t0DebtRemaining_ =  borrower.t0Debt;
    t0DebtSettled_   -= t0DebtRemaining_;

    emit Settle(params_.borrower, t0DebtSettled_);

    if (borrower.t0Debt == 0) {
        // settle auction
```

```
        borrower.collateral = _settleAuction(
            auctions_,
            buckets_,
            deposits_,
            params_.borrower,
            borrower.collateral,
            params_.poolType
        );
    }

    collateralRemaining_ =  borrower.collateral;
    collateralSettled_    -= collateralRemaining_;

    // update borrower state
    loans_.borrowers[params_.borrower] = borrower;
}
```

Tool used

Manual Review

Recommendation

Consider rebalancing each time when there is something to rebalance, for example:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L353-L385

```
    function settle(
        address borrowerAddress_,
        uint256 maxDepth_
    ) external nonReentrant override {
        ...
        (
            uint256 collateralRemaining,
            uint256 t0DebtRemaining,
            uint256 collateralSettled,
            uint256 t0DebtSettled
        ) = Auctions.settlePoolDebt(
            ...
        );

        // slither-disable-next-line incorrect-equality
-        if (t0DebtRemaining == 0) _rebalanceTokens(params.borrower,
↪   collateralRemaining);
```

```
+         if (collateralSettled > 0) _rebalanceTokens(params.borrower,
↪   collateralRemaining);
```

Issue M-11: Deposits are eliminated before currently unclaimed reserves when there is no reserve auction

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/102

Found by

hyh

Summary

Reserves that were unclaimed during last reserve auction that's now ended are not utilized for bad debt coverage and are treated as liabilities despite it is the free reserve funds of the pool.

Due to that deposits are being written off when there are still reserve funds exist and deposits' turn as a last resort liquidity source aren't came yet.

Vulnerability Detail

Suppose auctioned reserves weren't taken for any reason: say no market participants were there for that particular pool in the period when reserve auction implied Ajna token price was above market. Then there is no liability, i.e. that amount is free pool funds and to be used ahead of HPB deposits to cover any deficits.

Currently that's not happening, instead unclaimed reserves are frozen and aren't used. I.e. system treats these funds as being liable (while they aren't, auction is ended), so only very last reserve funds, that weren't yet added to the reserve auctions pot, can be used to cover bad debt. When there are not enough such funds, deposits are written off.

Impact

Deposit holders take a loss when the pool in fact do have reserve funds to cover bad debt. This loss isn't a part of the declared mechanics of the protocol.

Reserve auction can end up with not all auctioned reserves taken frequently enough due to, for example:

- short period of time when Ajna token were overpriced in it,
- or this period intersecting with spike of gas prices that made it unprofitable in absolute terms,
- or low liquidity of Ajna token at that time.

I.e. the reason can vary, the point is reserve auction not being sold out can be a regular outcome, while the expected sequence of funds to cover bad debt is typical and is stated in whitepaper (part 7.6, Settling, point 3 in the list), so it will be expected by the lenders that the reserves are covering bad debt first.

Code Snippet

settlePoolDebt() uses the reserves to cover bad debt:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L277-L313

```
// if there's still debt and no collateral
if (borrower.t0Debt != 0 && borrower.collateral == 0) {
    // settle debt from reserves -- round reserves down however
    borrower.t0Debt -= Maths.min(borrower.t0Debt, (params_.reserves /
↪   params_.inflator) * 1e18);

    // if there's still debt after settling from reserves then start to forgive
↪   amount from next HPB
    // loop through remaining buckets if there's still debt to settle
    while (params_.bucketDepth != 0 && borrower.t0Debt != 0) {
        SettleLocalVars memory vars;

        (vars.index, , vars.scale) = Deposits.findIndexAndSumOfSum(deposits_, 1);
        vars.unscaledDeposit = Deposits.unscaledValueAt(deposits_, vars.index);
        vars.depositToRemove = Maths.wmul(vars.scale, vars.unscaledDeposit);
        vars.debt            = Maths.wmul(borrower.t0Debt, params_.inflator);

        // enough deposit in bucket to settle entire debt
        if (vars.depositToRemove >= vars.debt) {
            Deposits.unscaledRemove(deposits_, vars.index, Maths.wdiv(vars.debt,
↪   vars.scale));
            borrower.t0Debt  = 0;
↪                 // no remaining debt to settle

        // not enough deposit to settle entire debt, we settle only deposit
↪   amount
        } else {
            borrower.t0Debt -= Maths.wdiv(vars.depositToRemove,
↪   params_.inflator);             // subtract from remaining debt the
↪   corresponding t0 amount of deposit

            Deposits.unscaledRemove(deposits_, vars.index,
↪   vars.unscaledDeposit);              // Remove all deposit from bucket
            Bucket storage hpbBucket = buckets_[vars.index];
```

```
            if (hpbBucket.collateral == 0) {
↪                   // existing LPB and LP tokens for the bucket shall become
↪   unclaimable.
                emit BucketBankruptcy(vars.index, hpbBucket.lps);
                hpbBucket.lps            = 0;
                hpbBucket.bankruptcyTime = block.timestamp;
            }
        }

        --params_.bucketDepth;
    }
}
```

But this reserves do not include `reserveAuction.unclaimed`, which is treated like a liability even when there is no reserve auction:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L353-L365

```
function settle(
    address borrowerAddress_,
    uint256 maxDepth_
) external nonReentrant override {
    PoolState memory poolState = _accruePoolInterest();

    uint256 assets = Maths.wmul(poolBalances.t0Debt, poolState.inflator) +
↪   _getPoolQuoteTokenBalance();
    uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reserveAuction.unclaimed;

    SettleParams memory params = SettleParams(
        {
            borrower:    borrowerAddress_,
            reserves:    (assets > liabilities) ? (assets-liabilities) : 0,
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC20Pool.sol#L356-L378

```
function settle(
    address borrowerAddress_,
    uint256 maxDepth_
) external override nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

    uint256 assets = Maths.wmul(poolBalances.t0Debt, poolState.inflator) +
↪   _getPoolQuoteTokenBalance();
```

```
    uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reserveAuction.unclaimed;

    (
        ,
        ,
        uint256 collateralSettled,
        uint256 t0DebtSettled
    ) = Auctions.settlePoolDebt(
        auctions,
        buckets,
        deposits,
        loans,
        SettleParams({
            borrower:    borrowerAddress_,
            reserves:    (assets > liabilities) ? (assets - liabilities) : 0,
```

Reserve auction finishes by timer and there is no adjustments to `unclaimed` if it is not sold fully:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L642-L663

```
function takeReserves(
    ReserveAuctionState storage reserveAuction_,
    uint256 maxAmount_
) external returns (uint256 amount_, uint256 ajnaRequired_) {
    uint256 kicked = reserveAuction_.kicked;

    if (kicked != 0 && block.timestamp - kicked <= 72 hours) {
        uint256 unclaimed = reserveAuction_.unclaimed;
        uint256 price     = _reserveAuctionPrice(kicked);

        amount_       = Maths.min(unclaimed, maxAmount_);
        ajnaRequired_ = Maths.wmul(amount_, price);

        unclaimed -= amount_;

        reserveAuction_.unclaimed = unclaimed;

        emit ReserveAuction(unclaimed, price);
    } else {
        revert NoReservesAuction();
    }
}
```

Tool used

Manual Review

Recommendation

Consider removing currently unsettled `reserveAuction.unclaimed` if reserve auction doesn't take place now as those aren't liabilities:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L353-L382

```
    function settle(
        address borrowerAddress_,
        uint256 maxDepth_
    ) external nonReentrant override {
        PoolState memory poolState = _accruePoolInterest();
+       uint256 kicked = reserveAuction.kicked;
+       uint256 reservesAuctioned = (kicked != 0 && block.timestamp - kicked <=
↪   72 hours) ? reserveAuction.unclaimed : 0;

        uint256 assets = Maths.wmul(poolBalances.t0Debt, poolState.inflator) +
↪   _getPoolQuoteTokenBalance();
-       uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reserveAuction.unclaimed;
+       uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reservesAuctioned;

        SettleParams memory params = SettleParams(
            {
                borrower:    borrowerAddress_,
                reserves:    (assets > liabilities) ? (assets-liabilities) : 0,
                inflator:    poolState.inflator,
                bucketDepth: maxDepth_,
                poolType:    poolState.poolType
            }
        );
        (
            uint256 collateralRemaining,
            uint256 t0DebtRemaining,
            uint256 collateralSettled,
            uint256 t0DebtSettled
        ) = Auctions.settlePoolDebt(
            auctions,
            buckets,
            deposits,
            loans,
+           reserveAuction,
```

```
            params
        );
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC20Pool.sol#L356-L378

```
    function settle(
        address borrowerAddress_,
        uint256 maxDepth_
    ) external override nonReentrant {
        PoolState memory poolState = _accruePoolInterest();
+       uint256 kicked = reserveAuction.kicked;
+       uint256 reservesAuctioned = (kicked != 0 && block.timestamp - kicked <=
↪   72 hours) ? reserveAuction.unclaimed : 0;

        uint256 assets = Maths.wmul(poolBalances.t0Debt, poolState.inflator) +
↪   _getPoolQuoteTokenBalance();

-       uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reserveAuction.unclaimed;
+       uint256 liabilities = Deposits.treeSum(deposits) +
↪   auctions.totalBondEscrowed + reservesAuctioned;

        (
            ,
            ,
            uint256 collateralSettled,
            uint256 t0DebtSettled
        ) = Auctions.settlePoolDebt(
            auctions,
            buckets,
            deposits,
            loans,
+           reserveAuction,
            SettleParams({
                borrower:    borrowerAddress_,
                reserves:    (assets > liabilities) ? (assets - liabilities) : 0,
```

Reserve auction state can be added as an argument to provide these fields:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L199-L205

```
    function settlePoolDebt(
        AuctionsState storage auctions_,
        mapping(uint256 => Bucket) storage buckets_,
        DepositsState storage deposits_,
```

```
        LoansState storage loans_,
+       ReserveAuctionState storage reserveAuction_,
        SettleParams memory params_
    ) external returns (
```

Also, consider accounting for the reserves that were used to cover bad debt (otherwise next reserve auction will be frozen until new income replenishes the funds used for coverage):

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L277-L283

```
        // if there's still debt and no collateral
        if (borrower.t0Debt != 0 && borrower.collateral == 0) {
            // settle debt from reserves -- round reserves down however
+           uint256 reservesUsed = Maths.min(borrower.t0Debt, (params_.reserves
↪   / params_.inflator) * 1e18);
+           uint256 kicked       = reserveAuction_.kicked;
-           borrower.t0Debt -= Maths.min(borrower.t0Debt, (params_.reserves /
↪   params_.inflator) * 1e18);
+           borrower.t0Debt -= reservesUsed;
+           if (kicked != 0 && block.timestamp - kicked <= 72 hours)
↪   reserveAuction_.unclaimed -= reservesUsed;

            // if there's still debt after settling from reserves then start to
↪   forgive amount from next HPB
            // loop through remaining buckets if there's still debt to settle
```

Discussion

grandizzy

unclaimed auction reserves won't be available to fund a liquidation, this is an issue which we are going to document, no code change involved

hrishibhat

Classifying this issue as a medium based on the above comment.

Issue M-12: Flashloan end result isn't controlled

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/101

Found by

hyh, ctf_sec, minhtrng, CRYP70

Summary

FlashLoan logic do not control the end result of transferring tokens out and back in. Given that protocol aims to support arbitrary non-rebasing / without fee on transfer / decimals in `[1, 18]` fungible tokens to be quote and collateral of a pool, this includes any exotic types of behavior, for example, reporting a successful transfer, but not performing internal accounting update for any reason.

As an example, this can be a kind of wide blacklisting mechanics introduction (i.e. allow this white list of accounts, freeze everyone else type of logic).

Vulnerability Detail

Now there is no control of the resulting balance, and any token that successfully performs safeTransfer, but for any reason withholds an update of token internal accounting, can successfully steal the whole pool's balance of any Ajna pool. This can be initiated by an attacker unrelated to token itself as griefing.

As many core token contracts are upgradable (USDC, USDT and so forth), such behaviour can be not in place right now, but can be introduced in the future.

Impact

Some fungible tokens that qualify for Ajna pools (including not imposing any fee on transfers) may not return the whole amount back, but will report successful safeTransfer(), i.e. up to the whole balance of Ajna pool for such ERC20 token can be stolen.

This can take place in a situation when a popular token was upgraded and the consequences of the internal logic change weren't fully understood by wide market initially and most depositors remained in the corresponding Ajna pool, then someone calls a flash loan as a griefing attack that will result in the token freezing the balancer of the pool. Or it was understood, but the griefer was quicker.

As the probability of such internal mechanics introduction is low, but the impact is up to full loss of user's funds, setting the severity to be medium.

Code Snippet

Flash loan functions do not employ any checks after ERC20 token was received back:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC20Pool.sol#L236-L256

```
/// @inheritdoc FlashloanablePool
function flashLoan(
    IERC3156FlashBorrower receiver_,
    address token_,
    uint256 amount_,
    bytes calldata data_
) external override(IERC3156FlashLender, FlashloanablePool) nonReentrant returns
↪   (bool) {
    if (token_ == _getArgAddress(QUOTE_ADDRESS)) return
↪   _flashLoanQuoteToken(receiver_, token_, amount_, data_);

    if (token_ == _getArgAddress(COLLATERAL_ADDRESS)) {
        _transferCollateral(address(receiver_), amount_);

        if (receiver_.onFlashLoan(msg.sender, token_, amount_, 0, data_) !=
            keccak256("ERC3156FlashBorrower.onFlashLoan")) revert
↪   FlashloanCallbackFailed();

        _transferCollateralFrom(address(receiver_), amount_);
        return true;
    }

    revert FlashloanUnavailableForToken();
}
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/FlashloanablePool.sol#L33-L45

```
function _flashLoanQuoteToken(IERC3156FlashBorrower receiver_,
    address token_,
    uint256 amount_,
    bytes calldata data_
) internal returns (bool) {
    _transferQuoteToken(address(receiver_), amount_);

    if (receiver_.onFlashLoan(msg.sender, token_, amount_, 0, data_) !=
        keccak256("ERC3156FlashBorrower.onFlashLoan")) revert
↪   FlashloanCallbackFailed();

    _transferQuoteTokenFrom(address(receiver_), amount_);
```

```
    return true;
}
```

Flash loan safety is now controlled by safeTransfer() only, which internal mechanics can vary between ERC20 tokens:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L502-L508

```
function _transferQuoteTokenFrom(address from_, uint256 amount_) internal {
    IERC20(_getArgAddress(QUOTE_ADDRESS)).safeTransferFrom(from_, address(this),
↪   amount_ / _getArgUint256(QUOTE_SCALE));
}

function _transferQuoteToken(address to_, uint256 amount_) internal {
    IERC20(_getArgAddress(QUOTE_ADDRESS)).safeTransfer(to_, amount_ /
↪   _getArgUint256(QUOTE_SCALE));
}
```

Tool used

Manual Review

Recommendation

Consider adding a balance control check to ensure that flash loan invariant remains: record contract balance before `receiver_.onFlashLoan(...)` callback and record it after `_transferQuoteTokenFrom(address(receiver_), amount_)`, require that resulting token balance ends up being not less than initial.

This applies both to ERC20Pool's flashLoan() and FlashloanablePool's _flashLoanQuoteToken().

Issue M-13: Interest rates can be raised above the market as a griefing, disabling the pool

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/100

Found by

hyh

Summary

Interest rates algorithm is based on `MAU` to `TAU` dynamics, where TAU is 3.5 day EMA of total debt to LUP * total collateral. The latter value can be manipulated by a big collateral holder by becoming a borrower with insignificant debt and lots of collateral, so 3.5d EMA of `Debt / (LUP * Collateral)` will become depressed and rates will go up irrespective to the real debt supply/demand situation.

Vulnerability Detail

Let's suppose Bob is a big WBTC holder and current market rate for its lending is insignificant, say base WBTC deposit APY on major platforms is below 5 basis points. Say Bob is a big lender of USDC-WBTC (quote-collateral) pool or Bob has interests in disturbing that pool operations for any reasons, for example Bob is a beneficiary of a rival lending protocol.

Bob can borrow a minimal loan, say `1000 USDC` with big, magnitudes excessive, WBTC collateral, say `1000 WBTC`. As the pool is permissionless it is safe, Bob can withdraw any time, market risk is close to zero as the loan is too small, interest rates risk is small too as Bob left near zero market interest rate for the strict zero income while the WBTC is used as collateral in the pool. I.e. it's low risk, low cost strategy for Bob to do so.

Pool, on the other hand, will experience gradual rise of the interest rate as while `MAU` will stay relatively constant, `TAU` will become low due to total collateral amount being big (and stable, so EMA will move to the corresponding value), while other parts of `Debt / (LUP * Collateral)` be relatively constant.

Observing the rise of interest rates above market the borrowers will gradually leave. But not all, and Bob has achieved above market interest income from dormant part of the borrowers, who are slow to react to this dynamics. But, given borrowers being mostly rational and informed, this to be relatively short-term situation. More importantly, as the rate went up and borrowers has left, lenders will observe significantly decreased utilization and will leave pool as well, not receiving enough interest income for their deposits.

This way Bob essentially disturbed the USDC-WBTC pool, so he can leave some small part of WBTC collateral there so that the rate will stay elevated and pool remain to be unusable due to significantly elevated interest rate, as no borrower will enter there on such conditions.

Impact

Pool utility for market participants can be destroyed by manipulating the interest rate algorithm, so such pool becomes unusable and end up being abandoned. Since for a pair of quote-collateral there can be only one pool this effectively disturb the whole line of business, i.e. profit from say USDC quote, WBTC collateral operations will cease to exist for Ajna token holders.

Current borrowers can experience losses from the manipulated above market interest rate. Dormant borrowers, i.e. ones who be slow to react, will be hit the hardest.

Attack cost is proportional to the current risk-free market interest rate of the collateral as attacker gives it up for a while. This can be low enough for the majority of widely utilized collateral assets.

Code Snippet

Target utilization TAU is computed from average collateralization ratio, `Debt / Collateral`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/PoolCommons.sol#L56-L124

```
function updateInterestRate(
    InterestState storage interestParams_,
    DepositsState storage deposits_,
    PoolState memory poolState_,
    uint256 lup_
) external {

    // current values of EMA samples
    uint256 curDebtEma   = interestParams_.debtEma;
    uint256 curLupColEma = interestParams_.lupColEma;

    // meaningful actual utilization
    int256 mau;
    // meaningful actual utilization * 1.02
    int256 mau102;

    if (poolState_.debt != 0) {
        // update pool EMAs for target utilization calculation

        curDebtEma =
```

```
            Maths.wmul(poolState_.debt,  EMA_7D_RATE_FACTOR) +
            Maths.wmul(curDebtEma,       LAMBDA_EMA_7D
        );

        // lup * collateral EMA sample max value is 10 times current debt
        uint256 maxLupColEma = Maths.wmul(poolState_.debt, Maths.wad(10));

        // current lup * collateral value
        uint256 lupCol = Maths.wmul(poolState_.collateral, lup_);

        curLupColEma =
            Maths.wmul(Maths.min(lupCol, maxLupColEma), EMA_7D_RATE_FACTOR) +
            Maths.wmul(curLupColEma,                     LAMBDA_EMA_7D);

        // save EMA samples in storage
        interestParams_.debtEma   = curDebtEma;
        interestParams_.lupColEma = curLupColEma;

        // calculate meaningful actual utilization for interest rate update
        mau    = int256(_utilization(deposits_, poolState_.debt,
    ↪   poolState_.collateral));
        mau102 = mau * PERCENT_102 / 1e18;

    }

    // calculate target utilization
    int256 tu = (curDebtEma != 0 && curLupColEma != 0) ?
↪   int256(Maths.wdiv(curDebtEma, curLupColEma)) : int(Maths.WAD);

    if (!poolState_.isNewInterestAccrued) poolState_.rate =
↪   interestParams_.interestRate;

    uint256 newInterestRate = poolState_.rate;

    // raise rates if 4*(tu-1.02*mau) < (tu+1.02*mau-1)^2-1
    if (4 * (tu - mau102) < ((tu + mau102 - 1e18) ** 2) / 1e18 - 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);
    }

    newInterestRate = Maths.min(500 * 1e18, Maths.max(0.001 * 1e18,
↪   newInterestRate));

    if (poolState_.rate != newInterestRate) {
        interestParams_.interestRate       = uint208(newInterestRate);
```

```
        interestParams_.interestRateUpdate = uint48(block.timestamp);

        emit UpdateInterestRate(poolState_.rate, newInterestRate);
    }
}
```

_updateInterestState() and _accruePoolInterest() are called within all state changing operations of the pool:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L481-L488

```
function _updateInterestState(
    PoolState memory poolState_,
    uint256 lup_
) internal {
    // if it has been more than 12 hours since the last interest rate update,
 ↪  call updateInterestRate function
    if (block.timestamp - interestState.interestRateUpdate > 12 hours) {
        PoolCommons.updateInterestRate(interestState, deposits, poolState_,
 ↪  lup_);
    }
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L426-L431

```
function _accruePoolInterest() internal returns (PoolState memory poolState_) {
    // retrieve t0Debt amount from poolBalances struct
    uint256 t0Debt = poolBalances.t0Debt;

    // initialize fields of poolState_ struct with initial values
    poolState_.collateral     = poolBalances.pledgedCollateral;
```

Pool collateral is updated on any borrowing:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/BorrowerActions.sol#L100-L159

```
function drawDebt(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState    storage loans_,
    PoolState calldata poolState_,
    address borrowerAddress_,
    uint256 amountToBorrow_,
    uint256 limitIndex_,
```

```
    uint256 collateralToPledge_
) external returns (
    DrawDebtResult memory result_
) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    result_.poolDebt       = poolState_.debt;
    result_.newLup         = _lup(deposits_, result_.poolDebt);
    result_.poolCollateral = poolState_.collateral;

    ...

    // pledge collateral to pool
    if (collateralToPledge_ != 0) {
        // add new amount of collateral to pledge to borrower balance
        borrower.collateral  += collateralToPledge_;

        ...

        // add new amount of collateral to pledge to pool balance
        result_.poolCollateral += collateralToPledge_;
    }
```

Tool used

Manual Review

Recommendation

Per discussions so far the most effective approach, proposed by Matt, looks to be the weighting the collateral with the corresponding debt, i.e. instead of computing `sum(D_i) / sum(C_i)` (we omit `1 / LUP` term as it's constant here), which is the average collateralization ratio, the debt weighted version of it can be used, `sum(D_i ^ 2) / sum(C_i * D_i)`, which is the average collateralization ratio weighted by current debt.

As obtaining any significant debt brings in both market and interest rate risk, i.e. will raise the probability of attacker's borrow position liquidation and also ends up paying the elevated interest rate proportionally to the debt acquired, it will substantially raise the cost and diminish practical probability of the attack.

Issue M-14: Interest rate for pool is bounded wrongly

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/96

Found by

yixxas

Summary

It is documented that pools can be created for tokens with interest rate between 1-10%.

> Pool creators: create pool by providing a fungible token for quote and collateral and an interest rate between 1-10%

However, due to a wrong implementation, pools can only be created between 2-9%.

Vulnerability Detail

In PoolDeployer.sol contract we have `MIN_RATE = 0.01 * 1e18` and `MAX_RATE = 0.1 * 1e18`. This indicates the 1% and 10% value in which we should allow interest rate to be set.

However, in our `canDeploy` modifier, it causes a revert when the following condition is true.

```
if (MIN_RATE >= interestRate_ || interestRate_ >= MAX_RATE)
revert IPoolFactory.PoolInterestRateInvalid()
```

A more than or equal sign is used to do the comparison, and reverts. This means that we can only set interest rate in the range of 2-9%, which I believe is not intended.

Impact

Interest rate is bounded wrongly, limiting pools from being deployed with the intended range.

Code Snippet

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/PoolDeployer.sol#L13-L14 https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/PoolDeployer.sol#L38-L43

Tool used

Manual Review

Recommendation

Change to a strict comparison instead when doing the comparison.

```
- if (MIN_RATE >= interestRate_ || interestRate_ >= MAX_RATE)         revert
↪  IPoolFactory.PoolInterestRateInvalid();
+ if (MIN_RATE > interestRate_ || interestRate_ > MAX_RATE)         revert
↪  IPoolFactory.PoolInterestRateInvalid();
```

Issue M-15: Auction timers following liquidity can fall through the floor price causing pool insolvency

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/76

Found by

CRYP70

Summary

When a borrower cannot pay their debt in an ERC20 pool, their position is liquidated and their assets enter an auction for other users to purchase small pieces of their assets. Because of the incentive that users wish to not pay above the standard market price for a token, users will generally wait until assets on auction are as cheap as possible to purchase however, this is flawed because this guarantees a loss for all lenders participating in the protocol with each user that is liquidated.

Vulnerability Detail

Consider a situation where a user decides to short a coin through a loan and refuses to take the loss to retain the value of their position. When the auction is kicked off using the `kick()` function on this user, as time moves forward, the price for puchasing these assets becomes increasingly cheaper. These prices can fall through the floor price of the lending pool which will allow anybody to buy tokens for only a fraction of what they were worth originally leading to a state where the pool cant cover the debt of the user who has not paid their loan back with interest. The issue lies in the `_auctionPrice()` function of the `Auctions.sol` contract which calculates the price of the auctioned assets for the taker. This function does not consider the floor price of the pool. The proof of concept below outlines this scenario:

Proof of Concept:

```
function testInsolvency() public {

    // ============== Setup Scenario ==============
    uint256 interestRateOne = 0.05 * 10**18;            // Collateral  // Quote
↪   (loaned token, short position)
    address poolThreeAddr = erc20PoolFactory.deployPool(address(dai),
↪   address(weth), interestRateOne);
    ERC20Pool poolThree = ERC20Pool(address(poolThreeAddr));
    vm.label(poolThreeAddr, "DAI / WETH Pool Three");

    // Setup scenario and send liquidity providers some tokens
    vm.startPrank(address(daiDoner));
```

```
    dai.transfer(address(charlie), 3200 ether);
    vm.stopPrank();

    vm.startPrank(address(wethDoner));
    weth.transfer(address(bob), 1000 ether);
    vm.stopPrank();

    // ==============================================


    // Note At the time (24/01/2023) of writing ETH is currently 1,625.02 DAI,
    // so this would be a popular bucket to deposit in.

    // Start Scenario
    // The lower dowm we go the cheaper wETH becomes - At a concentrated fenwick index of 5635, 1 wETH = 1600 DAI (Approx real life price)
    uint256 fenwick = 5635;

    vm.startPrank(address(alice));
    weth.deposit{value: 2 ether}();
    weth.approve(address(poolThree), 2.226 ether);
    poolThree.addQuoteToken(2 ether, fenwick);
    vm.stopPrank();

    vm.startPrank(address(bob));
    weth.deposit{value: 9 ether}();
    weth.approve(address(poolThree), 9 ether);
    poolThree.addQuoteToken(9 ether, fenwick);
    vm.stopPrank();

    assertEq(weth.balanceOf(address(poolThree)), 11 ether);


    // ======================== start testing ========================

    vm.startPrank(address(bob));
    bytes32 poolSubsetHashes = keccak256("ERC20_NON_SUBSET_HASH");
    IPositionManagerOwnerActions.MintParams memory mp = IPositionManagerOwnerActions.MintParams({
        recipient: address(bob),
        pool: address(poolThree),
        poolSubsetHash: poolSubsetHashes
    });
    positionManager.mint(mp);
    positionManager.setApprovalForAll(address(rewardsManager), true);
    rewardsManager.stake(1);
    vm.stopPrank();
```

```
    assertEq(dai.balanceOf(address(charlie)), 3200 ether);
    vm.startPrank(address(charlie)); // Charlie runs away with the weth tokens
    dai.approve(address(poolThree), 3200 ether);
    poolThree.drawDebt(address(charlie), 2 ether, fenwick, 3200 ether);
    vm.stopPrank();

    vm.warp(block.timestamp + 62 days);


    vm.startPrank(address(bob));
    weth.deposit{value: 0.5 ether}();
    weth.approve(address(poolThree), 0.5 ether);
    poolThree.kick(address(charlie)); // Kick off liquidation
    vm.stopPrank();

    vm.warp(block.timestamp + 10 hours);

    assertEq(weth.balanceOf(address(poolThree)), 9020189981190878108); // 9 ether


    vm.startPrank(address(bob));
    // Bob Takes a (pretend) flashloan of 1000 weth to get cheap dai tokens
    weth.approve(address(poolThree), 1000 ether);
    poolThree.take(address(charlie), 1000 ether , address(bob), "");
    weth.approve(address(poolThree), 1000 ether);
    poolThree.take(address(charlie), 1000 ether , address(bob), "");
    weth.approve(address(poolThree), 1000 ether);
    poolThree.take(address(charlie), 1000 ether , address(bob), "");
    weth.approve(address(poolThree), 1000 ether);
    poolThree.take(address(charlie), 1000 ether, address(bob), "");

    poolThree.settle(address(charlie), 100);
    vm.stopPrank();


    assertEq(weth.balanceOf(address(poolThree)), 9152686732755985308); // Pool balance is still 9 ether instead of 11 ether - insolvency.
    assertEq(dai.balanceOf(address(bob)), 3200 ether); // The original amount that charlie posted as deposit


    vm.warp(block.timestamp + 2 hours);
    // users attempt to withdraw after shaken by a liquidation
    vm.startPrank(address(alice));
    poolThree.removeQuoteToken(2 ether, fenwick);
    vm.stopPrank();
```

```
    vm.startPrank(address(bob));
    poolThree.removeQuoteToken(9 ether, fenwick);
    vm.stopPrank();

    assertEq(weth.balanceOf(address(bob)), 1007664981389220443074); // 1007
↪   ether, originally 1009 ether
    assertEq(weth.balanceOf(address(alice)), 1626148471550317418); // 1.6 ether,
↪   originally 2 ether

}
```

Impact

An increase in borrowers who cant pay their debts back will result in a loss for all lenders.

Code Snippet

- https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/Auctions.sol#L1391-L1410

Tool used

Manual Review

Recommendation

It's recommended that the price of the assets on auction consider the fenwick(s) being used when determining the price of assets on loan and do not fall below that particular index. With this fix in place, the worst case scenario is that lenders can pruchase these assets for the price they were loaned out for allowing them to recover the loss.

Discussion

grandizzy

this is a design choice. however we're reconsidering the auction implementation to use a floor price

hrishibhat

Considering this issue a valid medium as there is a possible risk of funds lost for lenders under certain circumstances

Issue M-16: If borrower or kicker got blacklisted by asset contract their collateral or bond funds can be permanently frozen with the pool

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/75

Found by

hyh

Summary

It's impossible for borrower or kicker to transfer their otherwise withdraw-able funds to another address. If for some reason borrower or kicker got blacklisted by collateral or quote token contract (correspondingly), these funds will be permanently frozen as now there is no mechanics to move them to another address or specify the recipient for the transfer.

Vulnerability Detail

If during the duration of a loan the borrower got blacklisted by collateral asset contract, let's say it is USDC, there is no way to retrieve the collateral. These collateral funds will be permanently locked at the Pool contract balance.

Similar, although less dangerous as both duration and exposure is less, situation takes place with kicker's balance, that is referenced by `msg.sender` only and so bonds due will be frozen with the pool if that address be blacklisted.

For lender's case there is a position managing possibility via Pool's transferLPs() and PositionManager's memorializePositions() and reedemPositions(), but for a borrower and a kicker there is no way to transfer funds due ownership or even specify transfer recipient, so the corresponding collateral and bond funds will be frozen with the pool if current beneficiary be blacklisted.

Impact

Principal funds of borrower or kicker being permanently frozen in full, but backlisting is a low probability event, so setting the severity to be medium.

Code Snippet

When it is `vars.pull` BorrowerActions's repayDebt() require `borrowerAddress_ == msg.sender`:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/libraries/external/BorrowerActions.sol#L237-L330

```
function repayDebt(
    AuctionsState storage auctions_,
    mapping(uint256 => Bucket) storage buckets_,
    DepositsState storage deposits_,
    LoansState    storage loans_,
    PoolState calldata poolState_,
    address borrowerAddress_,
    uint256 maxQuoteTokenAmountToRepay_,
    uint256 collateralAmountToPull_
) external returns (
    RepayDebtResult memory result_
) {
    Borrower memory borrower = loans_.borrowers[borrowerAddress_];

    ...

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

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

        uint256 encumberedCollateral = borrower.t0Debt != 0 ?
↪   Maths.wdiv(vars.borrowerDebt, result_.newLup) : 0;

        if (borrower.collateral - encumberedCollateral <
↪   collateralAmountToPull_) revert InsufficientCollateral();

        // stamp borrower t0Np when pull collateral action
        vars.stampT0Np = true;

        borrower.collateral    -= collateralAmountToPull_;
        result_.poolCollateral -= collateralAmountToPull_;
    }
```

Then funds are being sent in ERC20Pool and ERC721Pool to `msg.sender` with no option to specify another address:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC20Pool.sol#L184-L230

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

```
    uint256 collateralAmountToPull_
) external nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

    ...
    collateralAmountToPull_     = _roundToScale(collateralAmountToPull_,
↪   _bucketCollateralDust(0));

    RepayDebtResult memory result = BorrowerActions.repayDebt(
        ...
    );

    ...
    if (collateralAmountToPull_ != 0) {
        // update pool balances state
        poolBalances.pledgedCollateral = result.poolCollateral;

        // move collateral from pool to sender
        _transferCollateral(msg.sender, collateralAmountToPull_);
    }
}
```

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L194-L238

```
function repayDebt(
    address borrowerAddress_,
    uint256 maxQuoteTokenAmountToRepay_,
    uint256 noOfNFTsToPull_
) external nonReentrant {
    PoolState memory poolState = _accruePoolInterest();

    RepayDebtResult memory result = BorrowerActions.repayDebt(
        ...
    );

    ...
    if (noOfNFTsToPull_ != 0) {
        // update pool balances state
        poolBalances.pledgedCollateral = result.poolCollateral;

        // move collateral from pool to sender
        _transferFromPoolToAddress(msg.sender, borrowerTokenIds[msg.sender],
↪   noOfNFTsToPull_);
    }
}
```

The same issue takes place in withdrawBonds() as `msg.sender` can be blacklisted in between kicking and withdrawing:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L318-L327

```
/**
 *  @inheritdoc IPoolLiquidationActions
 *  @dev write state:
 *       - reset kicker's claimable accumulator
 */
function withdrawBonds() external {
    uint256 claimable = auctions.kickers[msg.sender].claimable;
    auctions.kickers[msg.sender].claimable = 0;
    _transferQuoteToken(msg.sender, claimable);
}
```

I.e. if as of time of kicking `msg.sender` wasn't blacklisted, but they were added to blacklist before `auctions.kickers[msg.sender].claimable` were withdrawn, it will be permanently locked on the Pool's balance.

Tool used

Manual Review

Recommendation

Consider adding the `recipient` argument to the repayDebt() and withdrawBonds() functions, so the balance beneficiary `msg.sender` can specify what address should receive the funds, for example:

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/base/Pool.sol#L318-L327

```
    /**
     *  @inheritdoc IPoolLiquidationActions
     *  @dev write state:
     *       - reset kicker's claimable accumulator
     */
-   function withdrawBonds() external {
+   function withdrawBonds(address recipient) external {
        uint256 claimable = auctions.kickers[msg.sender].claimable;
        auctions.kickers[msg.sender].claimable = 0;
-       _transferQuoteToken(msg.sender, claimable);
+       _transferQuoteToken(recipient, claimable);
    }
```

This will also help for the situation when NFT collateral was put in by a contract borrower without `onERC721Received` implementation, so repayDebt() initiated `_transferNFT() -> safeTransferFrom()` call will fail for `msg.sender` and the ability to use a recipient become crucial.

Issue M-17: user can drawDebt that is below dust amount

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/70

Found by

koxuan

Summary

According to the protocol, drawDebt prevents user from drawing below the `quoteDust_` amount. However, a logical error in the code can allow user to draw below dust amount.

Vulnerability Detail

`_revertOnMinDebt` is used in `drawDebt` to prevent dust loans. As you can see, the protocol wants to take the average of debt in the pool and make it the minimum if there are 10 or more loans. If it is lower than 10 loans, a `quoteDust` is used as the minimum. There is an edge case, whereby there are 10 loans in the pool, and the borrowers repay the loans till there is only 1 unit owed for each loan.(Might revert due to rounding error but it is describing a situation whereby repaying till a low amount of poolDebt can enable this). A new borrower can then `drawDebt` and because `_revertOnMindebt` only goes through the average loan amount check and not the `quoteDust_` amount check, he/she is able to draw loan that is well below the `quoteDust_` amount.

```
function _revertOnMinDebt(
    LoansState storage loans_,
    uint256 poolDebt_,
    uint256 borrowerDebt_,
    uint256 quoteDust_
) view {
    if (borrowerDebt_ != 0) {
        uint256 loansCount = Loans.noOfLoans(loans_);
        if (loansCount >= 10) {
            if (borrowerDebt_ < _minDebtAmount(poolDebt_, loansCount)) revert
↪   AmountLTMinDebt();
        } else {
            if (borrowerDebt_ < quoteDust_)                              revert
↪   DustAmountNotExceeded();
        }
    }
}
```

```
function _minDebtAmount(
    uint256 debt_,
    uint256 loansCount_
) pure returns (uint256 minDebtAmount_) {
    if (loansCount_ != 0) {
        minDebtAmount_ = Maths.wdiv(Maths.wdiv(debt_, Maths.wad(loansCount_)),
↪   10**19);
    }
}
```

Impact

A minimum loan amount is used to deter dust loans, which can diminish user experience.

Code Snippet

BorrowerActions.sol#173 RevertsHelper.sol#L61-L75 PoolHelper.sol#L100-L107

Tool used

Manual Review

Recommendation

Recommend checking that loan amount is more than `quoteDust_` regardless of the loan count.

```
function _revertOnMinDebt(
    LoansState storage loans_,
    uint256 poolDebt_,
    uint256 borrowerDebt_,
    uint256 quoteDust_
) view {
    if (borrowerDebt_ != 0) {
        uint256 loansCount = Loans.noOfLoans(loans_);
        if (loansCount >= 10) {
            if (borrowerDebt_ < _minDebtAmount(poolDebt_, loansCount)) revert
↪   AmountLTMinDebt();
        }
        if (borrowerDebt_ < quoteDust_) revert DustAmountNotExceeded();

    }
```

```
}
```

Discussion

EdNoepel

Edge case discussed in the fourth sentence is invalid. Repayment performs a check to ensure remaining debt is also above the minimum debt amount (or 0). ~~As such, there is no way for `_minDebtAmount` to return a number smaller than the dust amount.~~

However, since `_minDebtAmount` is 10% of the average, one could game the system. Upon pool creation, the attacker creates 10 EOAs, and draws debt at the dust limit from each of them. Now the pool is in a state where the minimum debt is 10% of the dust limit.

I do not see how this would give the attacker a material advantage, but agree it should be remedied regardless.

Issue M-18: Quadratic voting tally done wrong

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/64

Found by

cducrest-brainbot

Summary

Quadratic voting tally is done wrong, which results in a risk of hijacking the grant distribution with relatively low centralization/control over tokens.

Vulnerability Detail

Regarding grant coordination voting of the funding stage, the technical spec states (page24):

> Each address that holds Ajna tokens can vote as much as they want on every proposal, including negative votes, subject to the constraint that the sum of the squares of their votes cannot exceed the square of the number of Ajna tokens held.

The code enforces that the sum of all votes of a user (in absolute value) is lower or equal than the square of its token holdings. The votes are not squared before being summed.

Impact

This leads to a higher centralization/control risk than should be. Alice can deploy a smart contract that proposes a funding of all the available tokens for that round towards itself. The contract can reward people delegating to it in case of a successful funding.

Due to the incorrect tally, if 100 token holders each have 1 token, Alice only needs to convince 10 of them to join her cause to successfully gain the funds. She would receive `10^2 = 100` votes, while all other voters could only produce `90 * 1^2 = 90` votes.

If the technical spec were respected and with the same token distribution of 100 tokens split among 100 holders, Alice would need to bribe 50 token holders to join her cause to gain the funding. She has `50^2` voting power and can vote once with`50` while all other can vote `50 * 1^2 = 50`.

I consider this difference significant enough to be considered medium severity.

Code Snippet

In `StandardFunding.sol` in `_fundingVote()` the votes for a proposal are counted by adding `budgetAllocation_`: https://github.com/sherlock-audit/2023-01-ajna/blob/main/ecosystem-coordination/src/grants/base/StandardFunding.sol#L373

This budget allocation is withdrawn from the voter's voting budget: https://github.com/sherlock-audit/2023-01-ajna/blob/main/ecosystem-coordination/src/grants/base/StandardFunding.sol#L358-L368

The initial voting budget is the square of the token holdings at past snapshot: https://github.com/sherlock-audit/2023-01-ajna/blob/main/ecosystem-coordination/src/grants/GrantFund.sol#L126-L141

Tool used

Manual Review

Testing with one person holding 2 tokens beating three persons holding 1 token each:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.16;

import { IGovernor } from "@oz/governance/IGovernor.sol";
import { IVotes }    from "@oz/governance/utils/IVotes.sol";
import { SafeCast }  from "@oz/utils/math/SafeCast.sol";

import { Funding }          from "../src/grants/base/Funding.sol";
import { GrantFund }        from "../src/grants/GrantFund.sol";
import { IStandardFunding } from "../src/grants/interfaces/IStandardFunding.sol";
import { Maths }            from "../src/grants/libraries/Maths.sol";

import { GrantFundTestHelper } from "./utils/GrantFundTestHelper.sol";
import { IAjnaToken }          from "./utils/IAjnaToken.sol";

import { console } from "forge-std/console.sol";

contract StandardFundingGrantFundTest is GrantFundTestHelper {

    // used to cast 256 to uint64 to match emit expectations
    using SafeCast for uint256;

    IAjnaToken        internal  _token;
    IVotes            internal  _votingToken;
    GrantFund         internal  _grantFund;

    // Ajna token Holder at the Ajna contract creation on mainnet
```

```
    address internal _tokenDeployer  =
↪   0x666cf594fB18622e1ddB91468309a7E194ccb799;
    address internal _tokenHolder1   = makeAddr("_tokenHolder1");
    address internal _tokenHolder2   = makeAddr("_tokenHolder2");
    address internal _tokenHolder3   = makeAddr("_tokenHolder3");
    address internal _tokenHolder4   = makeAddr("_tokenHolder4");

    address[] internal _votersArr = [
        _tokenHolder1,
        _tokenHolder2,
        _tokenHolder3,
        _tokenHolder4
    ];

    uint256 _initialAjnaTokenSupply   = 2_000_000_000 * 1e18;

    // at this block on mainnet, all ajna tokens belongs to _tokenDeployer
    uint256 internal _startBlock      = 16354861;

    mapping (uint256 => uint256) internal noOfVotesOnProposal;
    uint256[] internal topTenProposalIds;
    uint256[] internal potentialProposalsSlate;
    uint256 treasury = 500_000_000 * 1e18;

    function setUp() external {
        vm.createSelectFork(vm.envString("ETH_RPC_URL"), _startBlock);

        vm.startPrank(_tokenDeployer);

        // Ajna Token contract address on mainnet
        _token = IAjnaToken(0x9a96ec9B57Fb64FbC60B423d1f4da7691Bd35079);

        // deploy voting token wrapper
        _votingToken = IVotes(address(_token));

        // deploy growth fund contract
        _grantFund = new GrantFund(_votingToken, treasury);

        // initial minter distributes tokens to test addresses
        // _transferAjnaTokens(_token, _votersArr, 50_000_000 * 1e18,
↪   _tokenDeployer);
        changePrank(_tokenDeployer);
        _token.transfer(_tokenHolder1, 2 * 1e18);
        _token.transfer(_tokenHolder2, 1 * 1e18);
        _token.transfer(_tokenHolder3, 1 * 1e18);
        _token.transfer(_tokenHolder4, 1 * 1e18);

        // initial minter distributes treasury to grantFund
```

```
        _token.transfer(address(_grantFund), treasury);
    }

    /*************/
    /*** Tests ***/
    /*************/

    function testQuadraticVotingTally() external {
        _selfDelegateVoters(_token, _votersArr);

        vm.roll(_startBlock + 50);

        // start distribution period
        _startDistributionPeriod(_grantFund);
        uint256 distributionId = _grantFund.getDistributionId();
        (, , , , uint256 gbc, ) =
↪   _grantFund.getDistributionPeriodInfo(distributionId);

        // generate proposal targets
        address[] memory ajnaTokenTargets = new address[](1);
        ajnaTokenTargets[0] = address(_token);

        // generate proposal values
        uint256[] memory values = new uint256[](1);
        values[0] = 0;

        // generate proposal calldata
        bytes[] memory proposalCalldata = new bytes[](1);
        proposalCalldata[0] = abi.encodeWithSignature(
            "transfer(address,uint256)",
            _tokenHolder1,
            gbc * 8/10
        );
        bytes[] memory proposalCalldata2 = new bytes[](1);
        proposalCalldata2[0] = abi.encodeWithSignature(
            "transfer(address,uint256)",
            _tokenHolder2,
            gbc * 7/10
        );

        // create and submit proposal
        TestProposal memory proposal = _createProposalStandard(_grantFund,
↪   _tokenHolder1, ajnaTokenTargets, values, proposalCalldata, "Proposal for
↪   Ajna token transfer to tester address");
        TestProposal memory proposal2 = _createProposalStandard(_grantFund,
↪   _tokenHolder2, ajnaTokenTargets, values, proposalCalldata2, "Proposal 2 for
↪   Ajna token transfer to tester address");
```

```
        vm.roll(_startBlock + 200);

        // screening period votes
        _vote(_grantFund, _tokenHolder1, proposal.proposalId, voteYes, 1);
        _vote(_grantFund, _tokenHolder2, proposal2.proposalId, voteYes, 1);

        // skip forward to the funding stage
        vm.roll(_startBlock + 600_000);

        GrantFund.Proposal[] memory screenedProposals =
↪   _getProposalListFromProposalIds(_grantFund,
↪   _grantFund.getTopTenProposals(distributionId));
        assertEq(screenedProposals.length, 2);
        assertEq(screenedProposals[0].proposalId, proposal.proposalId);
        assertEq(screenedProposals[0].votesReceived, 2 * 1e18);
        assertEq(screenedProposals[1].proposalId, proposal2.proposalId);
        assertEq(screenedProposals[1].votesReceived, 1 * 1e18);

        // check initial voting power
        uint256 votingPower = _grantFund.getVotesWithParams(_tokenHolder1,
↪   block.number, "Funding");
        assertEq(votingPower, 4 * 1e18);
        votingPower = _grantFund.getVotesWithParams(_tokenHolder2, block.number,
↪   "Funding");
        assertEq(votingPower, 1 * 1e18);
        votingPower = _grantFund.getVotesWithParams(_tokenHolder3, block.number,
↪   "Funding");
        assertEq(votingPower, 1 * 1e18);
        votingPower = _grantFund.getVotesWithParams(_tokenHolder4, block.number,
↪   "Funding");
        assertEq(votingPower, 1 * 1e18);

        _fundingVote(_grantFund, _tokenHolder1, proposal.proposalId, voteYes, 4
↪   * 1e18);
        _fundingVote(_grantFund, _tokenHolder2, proposal2.proposalId, voteYes, 1
↪   * 1e18);
        _fundingVote(_grantFund, _tokenHolder3, proposal2.proposalId, voteYes, 1
↪   * 1e18);
        _fundingVote(_grantFund, _tokenHolder4, proposal2.proposalId, voteYes, 1
↪   * 1e18);

        // check voting power after voting
        votingPower = _grantFund.getVotesWithParams(_tokenHolder1, block.number,
↪   "Funding");
        assertEq(votingPower, 0 * 1e18);
        votingPower = _grantFund.getVotesWithParams(_tokenHolder2, block.number,
↪   "Funding");
        assertEq(votingPower, 0 * 1e18);
```

```
        votingPower = _grantFund.getVotesWithParams(_tokenHolder3, block.number,
↪   "Funding");
        assertEq(votingPower, 0 * 1e18);
        votingPower = _grantFund.getVotesWithParams(_tokenHolder4, block.number,
↪   "Funding");
        assertEq(votingPower, 0 * 1e18);

        // skip to the DistributionPeriod
        vm.roll(_startBlock + 650_000);

        uint256[] memory winningSlate = new uint256[](1);
        winningSlate[0] = proposal.proposalId;
        uint256[] memory losingSlate = new uint256[](1);
        losingSlate[0] = proposal2.proposalId;

        // The losing slate is valid
        assertTrue(_grantFund.checkSlate(losingSlate, distributionId));
        // The winning slate is valid and has more votes than the losing slate
        assertTrue(_grantFund.checkSlate(winningSlate, distributionId));
    }
}
```

Recommendation

Subtract the square of the votes from the voting budget when voting:

```
@@ -357,17 +361,10 @@ abstract contract StandardFunding is Funding,
↪   IStandardFunding {
         // case where voter is voting against the proposal
         if (budgetAllocation_ < 0) {
             support = 0;
-
-            // update voter budget remaining
-            voter_.budgetRemaining += budgetAllocation_;
-        }
-        // voter is voting in support of the proposal
-        else {
-            // update voter budget remaining
-            voter_.budgetRemaining -= budgetAllocation_;
         }
+        voter_.budgetRemaining -=
↪   int256(Maths.wpow(uint256(Maths.abs(budgetAllocation_)), 2));
         // update total vote cast
         currentDistribution.quadraticVotesCast +=
↪   uint256(Maths.abs(budgetAllocation_));

         // update proposal vote tracking
```

```
        proposal_.qvBudgetAllocated += budgetAllocation_;
```

Issue M-19: CryptoKitty and CryptoFighter NFT can be paused, which block borrowing / repaying / liquidating action in the ERC721Pool when borrowers still forced to pay the compounding interest

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/34

Found by

ctf_sec

Summary

CryptoKitty and CryptoFighter NFT can be paused, which block borrowing / repaying / liquidating action in the ERC721Pool

Vulnerability Detail

In the current implementation in the factory contract and the pool contract, special logic is in-place to handle non-standard NFT such as crypto-kitty, crypto-figher or crypto punk.

In the factory contract:

```
NFTTypes nftType;
// CryptoPunks NFTs
if (collateral_ == 0xb47e3cd837dDF8e4c57F05d70Ab865de6e193BBB ) {
    nftType = NFTTypes.CRYPTOPUNKS;
}
// CryptoKitties and CryptoFighters NFTs
else if (collateral_ == 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d ||
↪  collateral_ ==  0x87d598064c736dd0C712D329aFCFAA0Ccc1921A1) {
    nftType = NFTTypes.CRYPTOKITTIES;
}
// All other NFTs that support the EIP721 standard
else {
    // Here 0x80ac58cd is the ERC721 interface Id
    // Neither a standard NFT nor a non-standard supported NFT(punk, kitty or
↪  fighter)
    try IERC165(collateral_).supportsInterface(0x80ac58cd) returns (bool
↪  supportsERC721Interface) {
        if (!supportsERC721Interface) revert NFTNotSupported();
    } catch {
        revert NFTNotSupported();
    }
```

```
    nftType = NFTTypes.STANDARD_ERC721;
}
```

And in ERC721Pool When handling ERC721 token transfer:

```
/**
 * @notice Helper function for transferring multiple NFT tokens from msg.sender
↪ to pool.
 * @notice Reverts in case token id is not supported by subset pool.
 * @param  poolTokens_ Array in pool that tracks NFT ids (could be tracking
↪ NFTs pledged by borrower or NFTs added by a lender in a specific bucket).
 * @param  tokenIds_   Array of NFT token ids to transfer from msg.sender to
↪ pool.
 */
function _transferFromSenderToPool(
    uint256[] storage poolTokens_,
    uint256[] calldata tokenIds_
) internal {
    bool subset   = _getArgUint256(SUBSET) != 0;
    uint8 nftType = _getArgUint8(NFT_TYPE);

    for (uint256 i = 0; i < tokenIds_.length;) {
        uint256 tokenId = tokenIds_[i];
        if (subset && !tokenIdsAllowed[tokenId]) revert OnlySubset();
        poolTokens_.push(tokenId);

        if (nftType == uint8(NFTTypes.STANDARD_ERC721)){
            _transferNFT(msg.sender, address(this), tokenId);
        }
        else if (nftType == uint8(NFTTypes.CRYPTOKITTIES)) {

↪ ICryptoKitties(_getArgAddress(COLLATERAL_ADDRESS)).transferFrom(msg.sender
↪ ,address(this), tokenId);
        }
        else{
            ICryptoPunks(_getArgAddress(COLLATERAL_ADDRESS)).buyPunk(tokenId);
        }

        unchecked { ++i; }
    }
}
```

and

```
uint8 nftType = _getArgUint8(NFT_TYPE);
```

```
for (uint256 i = 0; i < amountToRemove_;) {
    uint256 tokenId = poolTokens_[--noOfNFTsInPool]; // start with transferring
↪   the last token added in bucket
    poolTokens_.pop();

    if (nftType == uint8(NFTTypes.STANDARD_ERC721)){
        _transferNFT(address(this), toAddress_, tokenId);
    }
    else if (nftType == uint8(NFTTypes.CRYPTOKITTIES)) {
        ICryptoKitties(_getArgAddress(COLLATERAL_ADDRESS)).transfer(toAddress_,
↪   tokenId);
    }
    else {

↪   ICryptoPunks(_getArgAddress(COLLATERAL_ADDRESS)).transferPunk(toAddress_,
↪   tokenId);
    }

    tokensTransferred[i] = tokenId;

    unchecked { ++i; }
}
```

note if the NFT address is classified as either crypto kitties or crypto fighers, then the NFT type is classified as CryptoKitties, then transfer and transferFrom method is triggered.

```
if (nftType == uint8(NFTTypes.CRYPTOKITTIES)) {

↪   ICryptoKitties(_getArgAddress(COLLATERAL_ADDRESS)).transferFrom(msg.sender
↪   ,address(this), tokenId);
        }
```

and

```
else if (nftType == uint8(NFTTypes.CRYPTOKITTIES)) {
    ICryptoKitties(_getArgAddress(COLLATERAL_ADDRESS)).transfer(toAddress_,
↪   tokenId);
}
```

However, in both crypto-kitty and in crypto-figher NFT, the transfer and transferFrom method can be paused.

In crypto-figher NFT:

https://etherscan.io/address/0x87d598064c736dd0C712D329aFCFAA0Ccc1921A1#code#L873

```
function transferFrom(
    address _from,
    address _to,
    uint256 _tokenId
)
    public
    whenNotPaused
{
```

In Crypto-kitty NFT:

https://etherscan.io/address/0x06012c8cf97BEaD5deAe237070F9587f8E7A266d#code#L615

```
function transferFrom(
    address _from,
    address _to,
    uint256 _tokenId
)
    external
    whenNotPaused
{
```

note the WhenNotPaused modifier.

Impact

If the transfer and transferFrom is paused in CryptoKitty and CryptoFighter NFT, the borrowing and repaying and liquidating action is blocked in ERC721Pool, the user cannot fully clear his debt and has to pay the compounding interest when the transfer is paused.

Code Snippet

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721PoolFactory.sol#L58-L80

https://github.com/sherlock-audit/2023-01-ajna/blob/main/contracts/src/ERC721Pool.sol#L591-L623

Tool used

Manual Review

Recommendation

Interest should not be charged when external contract is paused to borrower when the external contract pause the transfer and transferFrom.

Discussion

grandizzy

we're going to remove support for those NFTs and support only wrapped NFTs - will be fixed with the same fix as for #163 #140 #31

Issue M-20: Minting an NFT with a position on the same bucket as a previously minted NFT changes its deposit time

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/19

Found by

MalfurionWhitehat

Summary

Minting an NFT with a position on the same bucket as a previously minted NFT changes its deposit time.

Vulnerability Detail

This issue happens because, after a position is memorialized on the `PositionManager`, this contract will centralize LP positions from different users, but these will be mapped to the same address from the point of view of Ajna pools (different users will be mapped as the same `lender` from the point of view of a `Pool`).

If more than one user has memorialized a position to the same bucket index, LenderActions.transferLPs will update the `depositTime` to the maximum of the previous and new lender's values.

As a result, when a second user memorializes their position as an NFT, the first user's `depositTime` will be overwritten by the second user's (greater) `depositTime`.

Impact

The `depositTime` is used when applying early withdrawal fee and on bankruptcy LP calculation. One of the impacts of this issue is that a user might incur in withdrawal fees because of another user. See the following scenario:

1. User1 memorializes position on bucket B
2. One day passes, so User1 should not be affected by early withdrawal penalty
3. User2 memorializes position on bucket B, and `depositTime` for both users (as they are using `PositionManager`) is updated to this new value
4. User1 withdraws, will incur in early withdrawal penalty

Code Snippet

```
newLender.depositTime = Maths.max(lenderDepositTime, newLender.depositTime);
```

Tool used

Manual Review

Recommendation

Reconsider the `PositionsManager` architecture and the way the LP properties are stored on the Ajna protocol. Since positions are not tokenized by default, but instead are stored on state variables on each pool, it is arguably more complex to reason about transferring positions between two users. Conversely, the ERC-721 standard already provides straightforward methods for minting/burning/sending tokens for accounts. This issue could be fixed by removing the optionality of NFT minting through another independent contract (the `PositionManager.sol`), as it is centralizing other user's LPs, and instead to refactoring the Ajna pools so that every position is an ERC-721 NFT minted directly by the `Pool.sol`.

Issue M-21: Memorializing an NFT position on the same bucket of a previously memorialized NFT locks redemption

Source: https://github.com/sherlock-audit/2023-01-ajna-judging/issues/13

Found by

MalfurionWhitehat

Summary

Memorializing a position as an NFT on the same bucket of an existing memorialized position will not allow any of the owners to directly redeem it back later.

Vulnerability Detail

This issue happens because, after a position is memorialized on the `PositionManager`, this contract will centralize LP positions from different users, but these will be mapped to the same address from the point of view of Ajna pools (different users will be mapped as the same `lender` from the point of view of a `Pool`).

If more than one user has memorialized a position to the same bucket index, when attempting to `PositionManager.redeemPositions`, the call to pool.transferLPs will revert with NoAllowance, as `LenderActions` does not allow a transfer with value lower than the total `lenderLpBalance`.

Because of that, any of the users' that share a bucket `redeemPositions` calls will fail.

Impact

Although users that share a bucket with memorialized positions are not able to direct redeem their positions, they can eventually get their LPs back with a specific set of actions.

By first calling `PositionManager.moveLiquidity` to a bucket *without any other LPs* managed by `PositionsManager` (since `LenderActions.moveQuoteToken` accepts moving less liquidity than the total a LP balance for a specific bucket), and then calling `PositionManager.redeemPositions`, LP-ers will be able to redeem their positions back. Because of this possibility, this is a Medium-severity issue.

Code Snippet

```
diff --git a/contracts/tests/forge/PositionManager.t.sol
↪ b/contracts/tests/forge/PositionManager.t.sol
index 5e691fc..5d5e822 100644
--- a/contracts/tests/forge/PositionManager.t.sol
+++ b/contracts/tests/forge/PositionManager.t.sol
@@ -186,6 +186,91 @@ contract PositionManagerERC20PoolTest is
↪ PositionManagerERC20PoolHelperContract
         assertTrue(_positionManager.isIndexInPosition(tokenId, 2552));
     }

+    function testMemorializePositionsTwoAccountsSameBucket() external {
+        address alice = makeAddr("alice");
+        address bob = makeAddr("bob");
+        uint256 mintAmount  = 10_000 * 1e18;
+
+        uint256 lpBalance;
+        uint256 depositTime;
+
+        _mintQuoteAndApproveManagerTokens(alice, mintAmount);
+        _mintQuoteAndApproveManagerTokens(bob, mintAmount);
+
+        // call pool contract directly to add quote tokens
+        uint256[] memory indexes = new uint256[](1);
+        indexes[0] = 2550;
+
+        // alice adds liquidity now
+        _addInitialLiquidity(
+            {
+                from:   alice,
+                amount: 3_000 * 1e18,
+                index:  indexes[0]
+            }
+        );
+        (lpBalance, depositTime) = _pool.lenderInfo(indexes[0], alice);
+        uint256 aliceDepositTime = block.timestamp;
+        assertEq(lpBalance, 3_000 * 1e27);
+        assertEq(depositTime, aliceDepositTime);
+
+        // bob adds liquidity later
+        skip(1 hours);
+        _addInitialLiquidity(
+            {
+                from:   bob,
+                amount: 3_000 * 1e18,
+                index:  indexes[0]
```

```
+            }
+        );
+        (lpBalance, depositTime) = _pool.lenderInfo(indexes[0], bob);
+        assertEq(lpBalance, 3_000 * 1e27);
+        assertEq(depositTime, aliceDepositTime + 1 hours);
+
+
+        // bob memorializes first, alice memorializes second
+        address[] memory addresses = new address[](2);
+        addresses[0] = bob;
+        addresses[1] = alice;
+        uint256[] memory tokenIds = new uint256[](2);
+
+        // bob and alice mint an NFT to later memorialize existing positions
↪   into
+        tokenIds[0] = _mintNFT(bob, bob, address(_pool));
+        assertFalse(_positionManager.isIndexInPosition(tokenIds[0], 2550));
+        tokenIds[1] = _mintNFT(alice, alice, address(_pool));
+        assertFalse(_positionManager.isIndexInPosition(tokenIds[1], 2550));
+
+        for(uint256 i = 0; i < addresses.length; ++i) {
+            // construct memorialize params struct
+            IPositionManagerOwnerActions.MemorializePositionsParams memory
↪   memorializeParams = IPositionManagerOwnerActions.MemorializePositionsParams(
+                tokenIds[i], indexes
+            );
+
+            // allow position manager to take ownership of the position
+            changePrank(addresses[i]);
+            _pool.approveLpOwnership(address(_positionManager), indexes[0],
↪   3_000 * 1e27);
+
+            // memorialize quote tokens into minted NFT
+            vm.expectEmit(true, true, true, true);
+            emit MemorializePosition(addresses[i], tokenIds[i]);
+            vm.expectEmit(true, true, true, true);
+            emit TransferLPTokens(addresses[i], address(_positionManager),
↪   indexes, 3_000 * 1e27);
+
+            _positionManager.memorializePositions(memorializeParams);
+        }
+
+        // now both redeem
+        // will revert
+        for(uint256 i = 0; i < addresses.length; ++i) {
+            // construct memorialize params struct
+            IPositionManagerOwnerActions.RedeemPositionsParams memory params =
↪   IPositionManagerOwnerActions.RedeemPositionsParams(
```

```
+                 tokenIds[i], address(_pool), indexes
+             );
+             changePrank(addresses[i]);
+             _positionManager.reedemPositions(params);
+         }
+     }
+
      function testRememorializePositions() external {
          address testAddress = makeAddr("testAddress");
          uint256 mintAmount  = 50_000 * 1e18;
```

Tool used

Manual Review

Recommendation

There are some possible alternatives to solving this issue.

1. Allow `LenderActions.transferLPs` to transfer an amount different than the total LP balance. Appropriate care must be taken with approve/transfer, that do not exist now as approvals are always exact and cleared after every transfer, but they may become important if the this flow is refactored.
2. Reconsider the `PositionsManager` architecture and the way the LP balances are stored on the Ajna protocol. Since positions are not tokenized by default, but instead are stored on state variables on each pool, it is arguably more complex to reason about transferring positions between two users. Conversely, the ERC-20 and ERC-721 standards already provide straightforward methods for minting/burning/sending tokens for accounts. A suggestion is, for example, to remove the optionality of NFT minting through another independent contract (the `PositionManager.sol`), and instead to refactor the Ajna pools so that every position is an ERC-721 NFT minted directly by the `Pool.sol`.