



Security Assessment & Formal Verification Report



Ajna

December 2023

Prepared for
Ajna

Table of content

Project Summary	3
Project Scope	3
Protocol Overview	3
Findings Summary	4
Detailed Findings	6
H-01: A manipulation on LUP and HTP can drain the reserves in favor of an attacker	6
L-01: Calculation of the repay-amount in bucketTake() is not bounded as intended	7
L-02: A deposit take can lead to HTP crossing the LUP	9
L-03: MoveQuoteTokens can lose small amount of deposits, that pushes LUP below HTP in an edge cass.	9
Info-01: Lender may raise HTP to steal interest rewards from lender buckets under LUP	10
Info-02: An attacker can kick everyone in the system, forcing the lenders into a risk/gamble	11
Info-03: Minimal borrow amount mitigation could be avoided by first depositor	12
Info-04: Minimal borrow amount mitigation could be somewhat bypassed	13
Info-05: positionIndexes and positions aren't deleted when attempting a positionTokens deletion	13
Info-06: TakerActions.sol TakerLocalVars.factor is unused	13
Info-07: Use string.concat() or bytes.concat() instead of abi.encodePacked	14
Info-08: Default Visibility for constants	15
Info-09: Event is never emitted	16
Info-10: Change uint to uint256	16
Gas-01: State variables only set in the constructor should be declared immutable	17
Gas-02: uint256 to bool mapping: Utilizing Bitmaps to dramatically save on Gas	18
Gas-03: Increments/decrements can be unchecked in for-loops	19
Gas-04: ++i costs less gas compared to i++ or i += 1	21
Formal Verification	24
General Assumptions	24
Formal Verification Properties	26
KickerActions.sol	26
Loans.sol	27
Buckets.sol	27
LenderActions.sol	28
TakerActions.sol	29
BorrowerActions.sol	31
Disclaimer	32
About Certora	32

Project Summary

Project Scope

Repo Name	Repository	Commit	Compiler version	Platform
ajna-core	https://github.com/ajna-finance/ajna-core	dc4984	Solidity 0.8.18	EVM

Project Overview

This document describes the specification and verification of the **AJNA protocol** using the Certora Prover and manual code review findings. The work was undertaken from **16 October 2023** to **20 November 2023**.

The following contract list is included in our scope:

```
ajna-core/src/ERC20Pool.sol
ajna-core/src/ERC20PoolFactory.sol
ajna-core/src/ERC721Pool.sol
ajna-core/src/ERC721PoolFactory.sol
ajna-core/src/PoolInfoUtils.sol
ajna-core/src/PoolInfoUtilsMulticall.sol
ajna-core/src/PositionManager.sol
ajna-core/src/RewardsManager.sol
```

```
ajna-core/src/base/FlashloanablePool.sol
ajna-core/src/base/PermitERC721.sol
ajna-core/src/base/Pool.sol
ajna-core/src/base/PoolDeployer.sol
```

```
ajna-core/src/libraries/external/BorrowerActions.sol
ajna-core/src/libraries/external/KickerActions.sol
ajna-core/src/libraries/external/LenderActions.sol
```

```
ajna-core/src/libraries/external/LPActions.sol
ajna-core/src/libraries/external/PoolCommons.sol
ajna-core/src/libraries/external/PositionNFTSVG.sol
ajna-core/src/libraries/external/SettlerActions.sol
ajna-core/src/libraries/external/TakerActions.sol
```

```
ajna-core/src/libraries/helpers/PoolHelper.sol
ajna-core/src/libraries/helpers/RevertsHelper.sol
ajna-core/src/libraries/helpers/SafeTokenNamer.sol
```

```
ajna-core/src/libraries/internal/Buckets.sol
ajna-core/src/libraries/internal/Deposits.sol
ajna-core/src/libraries/internal/Loans.sol
ajna-core/src/libraries/internal/Maths.sol
```

The contracts were verified against Solidity version 0.8.18.

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts. During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	0	–	–
High	1	1	1
Medium	0	0	0
Low	3	2	1
Informational	10	–	–
Total	14	3	2

Detailed Findings

H-01: A manipulation on LUP and HTP can drain the reserves in favor of an attacker

Severity: high

Category: Drain of funds (partial)

File(s): N/A

Bug description: A sequence of operations that manipulates LUP and HTP, involving flashloan deposit, loan and kicking own loan at the rate of bucket 1, would drain most of the reserves in favor of an arbitrager (that can be the same attacker).

In addition, it seems the bond rewards calculation isn't linear, which may imply an error that could be amplifying the damage caused in this scenario.

Exploit scenario:

- Setup - there's a sane state where there's X amount of liquidity in Ajna, and Y amount of borrows.
- An attacker brings a lot of money atomically (flash-loan), and does this (with multiple addresses as needed):
 - Raised LUP via a deposit of $\sim Y$ to bucket 1.
 - Deposits Y amount into bucket 2 with a different address (attacker2).
 - Take a loan of $\sim Y$ at TP of bucket ~ 2 to raise the HTP.
 - Take another loan, at a safe TP to push the LUP back to the initial state.
 - Kick his own risky loan, which starts an auction and resets the HTP.
Note that Auction's params are very far off from the "real" market values.
 - Repays their loan completely -> raises LUP to 2
 - Withdraws their deposit from bucket 2 -> resets LUP to original value
- Thus far, the attacker brought $2Y$ and got almost $2Y$ back, at the cost of some collateral (in a TP of bucket ~ 2), and the Liquidation Bond (e.g. $\sim 0.022 Y$).
- The funds at bucket1 are frozen, as it's locked by the Auction (so no action is possible until it's resolved).
- The attacker (or any other MEVer /arbitrager) would close the auction at a price way higher than the real market. Then they immediately (same TX) withdraw what's left of bucket 1. They also withdraw the Bond rewards with Attacker3.
 - The attacker (and probably everyone) has an incentive to clear the Auction at way higher price than the real market price, as it opens the opportunity to gain what would be left of bucket 1.
 - By the time bucket1 is liquidated, it accrues a meaningful amount of interest.

- Although the lenders are relatively unaffected directly, the reserves (and claimable reserves) are substantially damaged (almost drained) compared to a scenario that elapses the same amount of time without the attacker's intervention.

Assuming that in that timeframe, market prices were stable. (it's unrelated to the effects of the above, but may incur more risk to the lenders as the reserves that should stave off bad debt are gone).

Ajna's response:

This issue is valid and was discovered in multiple audits. The outcome is that an attacker can effectively steal the reserves under certain conditions. Our fix is to limit the use of the reserves in the settle function to 50% of the loan's origination fee. We did not completely eliminate the use of the reserves in the settle function because if a malicious actor wished to grief other depositors in the pool, they could take out an enormous loan (incurring the origination fee at great expense to themselves) and allow the net interest margin to accrue as bad debt, which is then passed down to lower priced depositors. Since we know that the origination fee is one week of interest, and the maximum auction time frame is 72 hours, we can say with high confidence that 50% of the origination fee will cover any bad debt incurred from net interest margin in this specific scenario.

The issue was fixed in a later commit.

L-01: *Calculation of the repay-amount in bucketTake() is not bounded as intended.*

Severity: Low

Category: Unexpected underflow revert

File(s): TakerActions.sol

Property violated: Integrity of *calculateTakeFlowsAndBondChangeIntegrity()*

Bug description:

The [update](#) of `t0_debt` in `bucketTake()` in `TakerActions.sol` relies on the [computation of repay amount](#), which is (theoretically) supposed to be bounded by `t0_debt`, hence this line should never revert.

However, the round-down which [computes](#) `repayAmount` together with the computation of the [debt collateral constraint](#) can lead essentially to a higher value than intended, since the latter rounds up in general.

Here is a mini-rule implemented in CVL whose violation demonstrates the possibility of such a bug, purely mathematical:

Note: here we neglected the round-up-to-scale effect.

```
rule checkRoundingIssue(uint256 t0_debt, uint256 inflator, uint256 price) {
    require price !=0;
    require inflator == require_uint256( (WAD() * 101) / 100); /// 1%
    require t0_debt <= (1 << 160) && t0_debt >= 10;
    require price <= 10^30;
    uint256 repayAmount = mulDiv(
        ceilWdiv(wmul(t0_debt, inflator), price),
        price,
```

```
        inflator
    );
    assert repayAmount <= t0_debt;
}
```

Exploit scenario:

Example(1):

borrower price = 2356666666666666667 ~ 2.35 * WAD

inflator = 1.01 * WAD (1%)

t0_debt = 12

Hence: repayAmount = 14 > t0_debt.

Example(2):

borrower price = 0x53444835ec57f802 ~ 6 * WAD

inflator = 2 * WAD (100%)

t0_debt = 1e18 = WAD

Hence: repayAmount = WAD + 2 > t0_debt.

And these are only two examples out of many possibilities of counter-examples that do not satisfy this upper-bound limit. This, of course, shows that it is possible (in theory) to have an unexpected revert in the `bucketTake()` function.

The main problem with this bug is that it's unexpected and hard to predict. It's hard to say what should be the initial conditions of the borrower's debt and the buckets when this bug happens.

Obviously this is at most medium severity (also because of its frequency, but mainly because it can be avoided by waiting a block or two, or performing another operation).

Ajna's response:

We're curious to see if this can actually happen in practice. Regardless, we see the potential outcome and don't think it's worth changing any code. In the case where a user is paying off all of the debt, it should land in the middle of the if, else if, else clause in `_calculateTakeFlowsAndBondChange`.

The issue was fixed in a later commit.

L-02: A deposit take can lead to HTP crossing the LUP

Severity: Low

Category: Broken invariant

File(s): TakerActions.sol

Property violated: $HTP \leq LUP$.

Bug description: In a bucket take, as part of the auction mechanism, the amount of debt repaid is usually smaller than the amount that is removed from the bucket. In the case where the HTP is very close to the LUP, the difference between the bucket deposit removal and the debt repay amount can lead to a delta that would sharply change the LUP, crossing the HTP from above.

Exploit scenario: An auction of a kicked loan takes place during some time period, while some “bad borrower” decides to create a critical loan, whose TP is very close to the current LUP, making it the HTP.

During the auction of the kicked loan, some liquidators perform deposit take, which in general replace an amount of the bucket deposit quote tokens with the loan collateral, effectively reducing the amount of quote tokens that cover the existing loans. Because the penalty factor is larger than the kicker reward factor (or reward price is larger than the borrower price), then the amount of repaid debt could be small enough with respect to the bucket reduction, such that the LUP index changes abruptly (when on the edge of the bucket), making the LUP fall right below the HTP, thus breaking the invariant, in a non-borrow operation.

Ajna's response:

Acknowledged. The invariant is that a withdrawal of a deposit will never send the LUP below the HTP. The situation described does not violate an invariant and is an intentional part of the design. This is not a vulnerability.

L-03: MoveQuoteTokens can lose small amount of deposits, that pushes LUP below HTP in an edge case

Severity: Low

Category: Broken invariant, optimization misses edge case

File(s): LenderActions.sol

Property violated: $HTP \leq LUP$

Bug description: In the Fenwick tree's update, moving funds from one bucket to another might result in some amount lost due to rounding issues.

The rounding issue can cause diffs which can be more than 1 wei. It seems that the contributing factors are the difference between the two bucket scales, and the moved amount.

This can cause a lender action to break the central invariant.

In a scenario where the amount of deposits in all the buckets above the LUP index is very close to the total debt of the pool, this could push the LUP below the HTP.

Therefore, the check in [L#323-324](#), which means that the HTP doesn't surpass the LUP only if the funds moved downwards, seems like a logical optimization - but it misses this edge case that can also happen when the funds move upwards.

Exploit scenario: Given a market that had certain prices for a very prolonged time, and then moved slightly down - would introduce the scenario where there are two buckets over the LUP that have different inflators.

Suggestion: Consider removing said optimization, and always check that the LUP never crosses the HTP.

Ajna's response:

Acknowledged. In the event where this would occur, the LUP would most likely travel below the HTP anyway in the next block(s) because of interest accrual. So while this technically violates the invariant, it does not require a fix because it maintains the intent of the invariant, which is that a user cannot pass insufficiently collateralized debt down the book at other users' expense.

Info-01: Lender may raise HTP to steal interest rewards from lender buckets under LUP

Severity: Info

Category: Risk analysis - Edge case warps incentives

File(s): N/A

Issue description: According to design, lenders in the system are incentivized also to lend below the current LUP (although slightly punished), in order to gain equal interest rates. That's in order to encourage growth of the system.

The issue/edge case arises from the scenario that any lender can make a minimal borrow (10% of avg loan on normal circumstances) with the riskiest parameters possible, pushing HTP up to near LUP (the closer, the shorter the effect might be, but the more buckets harmed). The cost of this in the worst case scenario (where the borrower account is losing all its collateral) is just the delta between the market price and the LUP's position under it, in the form of a bit more collateral than market price.

Buckets that beforehand were in the buffer between the HTP and LUP would become ineligible for rewards, making the interest rates rewards of the buckets above the LUP higher (thus creating some incentive for a lender there to do that).

Furthermore, this could have several effects, like coercing those other lenders to move to a higher bucket, pressuring liquidity up to LUP, and possibly the LUP itself upwards (which might be repeated until market price, depending on liquidity spread); or making the other lenders go even further below the market price from the first place, hurting potential user base.

This loan also wouldn't allow the LUP to "naturally" fall below its current value, making the protocol needlessly more rigid.

Therefore, lenders that join the system at those rates below the LUP but above HTP, should consider whether it's far enough from market prices (and whether according to the liquidity's distribution) such an attack is possible against them (this can even be performed as a frontrun to their deposit). If yes, they should either join directly at LUP or stay out until LUP drops far enough from the market prices, such that it becomes unlikely.

Exploit scenario:

- Setup - there's a sane state (assumes LUP is market price) where there's X amount of liquidity in Ajna above the lup, and X_TAG of liquidity between LUP and HTP. There's a Y amount in loans which is smaller than X, and they're well backed such that X_TAG should be eligible as well.
- Any lender can come with another address, borrowing the minimal amount at a TP which is almost LUP.
- HTP is raised to LUP, and now the buckets that should have been rewarded for their liquidity don't get any rewards.
- The borrowed position is not kickable by any lender below LUP (or external entity), but the lenders above the LUP that can LenderKick wouldn't have the incentive to (unless they want to move down, but it wouldn't happen in stable prices or when there's enough liquidity at LUP). Thus this denial of rewards could be prolonged.

Ajna's response:

[Acknowledged.](#)

Info-02: An attacker can kick everyone in the system, forcing the lenders into a risk/gamble

Severity: Info

Category: Risk analysis - Edge case warps incentives

File(s): N/A

Issue description: According to design, lenders in the system should pick a price at which they're willing to lend. Furthermore, it's implied that they should take some buffer from real market prices since they might be locked by an auction.

The issue/edge case arises from the fact that there's a sequence of actions that allows anyone to kick anyone else in the system atomically - by decreasing the LUP via borrowing.

After the kick(s), said borrow could later be repaid, but all the lenders are frozen due to the auction - forcing them to take the "other side of the bet" until the auction prices drop to around market price levels (then the auctions would be cleared, and some funds would be unlocked).

Especially for volatile market prices, the chance of getting locked becomes more and more likely (as it may become more likely for someone to make this bet), thus increasing the risk.

The more volatile the market price is, the lower the price can get during the initial delay of an auction.

Therefore, with more volatility, the risk of losing value at a given price is higher, so the price they should consider “comfortable lending at” (and move their funds to) should consider this risk and be even further away from the current market prices. Or alternatively, be incentivised to quit the system beforehand..

Exploit scenario:

- Setup - there's a sane state where there's X amount of liquidity in Ajna, and Y amount of borrows.
- The attacker atomically:
 - Borrows $((X-Y) - \text{dust})$, in the riskiest TP possible -> HTP is just below LUP
 - Deposits at a bucket, lower than the safest borrow's TP (to target everyone in this example)
 - With a second address (Attacker2), borrows a “minimal_loan_size” - pushes LUP to their bucket
 - Kicks everyone in the system (their own initial borrow as well. A partial attack kicks only their own borrow)
 - Attacker2 repays -> LUP is pushed up to their bucket
 - withdraws their deposit
- Takers would come and call bucketTake() if prices were going below the original price (after the first ~6 hours).
- The lenders are left with just the collateral, which was devalued, and couldn't escape it in the few hours prior (which at least some could have, in the case without the attacker's intervention).

Ajna's response:

Acknowledged.

Info-03: Minimal borrow amount mitigation could be avoided by first depositor

Severity: Info

Category: Frontrun, griefing (to disincentivize other users)

File(s): -

Bug description: The minimal borrow amount is a mitigation against attacks raising HTP up to LUP without risk, causing the system to become more rigid than it should for the lenders.

This is only enforced after a certain amount of borrows exists, so either the first depositor (or any other depositor that acts before enough borrowers come) could try to do that in order to fix the HTP/LUP near a threshold they prefer, which could be riskier than other lenders would want.

This would undermine the incentives for lenders to lend under LUP in order to join the system, until this “grief” would be cleared.

Suggestion: Calculate the average utilized deposits instead of average borrow (e.g. the amount of QT lent above LUP (after the borrow), divided by the amount of lenders).

This way there is no need for a number of borrowers threshold, and also connect said incentive/promise directly to the lenders (still open to some manipulations, but to your consideration).

Ajna's response:

Acknowledged. This is a known trade-off that we chose not to change, we would categorize it as informational. The complexity of the code additions were not worth the gains.

Info-04: Minimal borrow amount mitigation could be somewhat bypassed

Severity: Info

Category: Lack of checks

File(s): -

Bug description: The minimal borrow amount is a mitigation against attacks raising HTP up to LUP without risk, causing the system to become more rigid than it should for the lenders.

This is calculated as a tenth of the average borrow size.

Beside the fact that the average borrow size seems disconnected from the promise we want to give the lenders, someone could open a lot of the minimal size positions atomically (also in decreasing sizes), dropping the average in order to take a small loan, then immediately repaying the others.

So the actual limit is not 1/10 of the avg position, but could be lower.

Suggestion: Have a state variable to remember this block's minimum loan size. This way this manipulation at least couldn't be atomic, introducing more possible risk to the attacker/griever.

Ajna's response:

Acknowledged. This is a known trade-off that we chose not to change. The complexity of the code additions were not worth the gains.

Info-05: positionIndexes and positions aren't deleted when attempting a positionTokens deletion

Severity: Info

Category: State handling

File(s): [PositionManager.sol](#)

Bug description: In [PositionManager.burn\(\)](#), there's an attempt to [delete positionTokens\[tokenId\]](#), which is a [TokenInfo struct](#). However, this struct contains [two mapping](#) members ([UintSet positionIndexes](#) and [mapping\(uint256 index => Position\) positions](#)) that aren't deleted recursively, therefore leaving behind the remaining after the function call and leaving them fetchable by view functions.

Ajna's response:

Acknowledged.

Info-06: *TakerActions.sol* *TakerLocalVars*.factor is unused

Severity: Informational

Category: unused variable

File(s): *TakerActions.sol*

Bug description: the *factor* field of *TakerLocalVars* is calculated first in *_prepareTake()* but isn't used anywhere.

Ajna's response:

Fixed in [this merge](#).

The issue was fixed in a later commit.

Info-07: Use `string.concat()` or `bytes.concat()` instead of `abi.encodePacked`

Severity: Informational

Category: Syntax

File(s): [ERC20PoolFactory.sol](#), [ERC721PoolFactory.sol](#), [PositionNFTSVG.sol](#)

Bug description:

Solidity version 0.8.4 introduces `bytes.concat()` (VS `abi.encodePacked(<bytes>, <bytes>)`)

Solidity version 0.8.12 introduces `string.concat()` (VS `abi.encodePacked(<str>, <str>)`), which catches concatenation errors (in the event of a bytes data mixed in the concatenation)

Affected code:

- [src/ERC20PoolFactory.sol](#)

File: `src/ERC20PoolFactory.sol`

```
ERC20PoolFactory.sol:61:      bytes memory data = abi.encodePacked(
```

- [src/ERC721PoolFactory.sol](#)

File: `src/ERC721PoolFactory.sol`

```
ERC721PoolFactory.sol:71:      bytes memory data = abi.encodePacked(
```

- [src/libraries/external/PositionNFTSVG.sol](#)

File: `src/libraries/external/PositionNFTSVG.sol`

```
PositionNFTSVG.sol:39:      abi.encodePacked("Ajna Token #",  
Strings.toString(params_.tokenId))
```

```
PositionNFTSVG.sol:45:      abi.encodePacked(
```

```
PositionNFTSVG.sol:49:      abi.encodePacked(
```

```
PositionNFTSVG.sol:73:          abi.encodePacked(
PositionNFTSVG.sol:85:      string memory backgroundTop = string(abi.encodePacked(
PositionNFTSVG.sol:98:      string memory backgroundMiddle = string(abi.encodePacked(
PositionNFTSVG.sol:110:     string memory backgroundBottom = string(abi.encodePacked(
PositionNFTSVG.sol:125:     background_ = string(abi.encodePacked(
PositionNFTSVG.sol:135:     defs_ = string(abi.encodePacked(
PositionNFTSVG.sol:155:     poolTag_ = string(abi.encodePacked(
PositionNFTSVG.sol:158:          abi.encodePacked(
PositionNFTSVG.sol:169:     tokenIdTag_ = string(abi.encodePacked(
```

Ajna's response:

[Acknowledged.](#)

Info-08: Default Visibility for constants

Severity: Informational

Category: Code clarity

File(s): [Loans.sol](#)

Bug description:

A constant is using the default visibility. For readability, consider explicitly declaring it as `internal`.

Affected code:

- [src/libraries/internal/Loans.sol](#)

File: `src/libraries/internal/Loans.sol`

```
Loans.sol:31:      uint256 constant ROOT_INDEX = 1;
```

Ajna's response:

[Acknowledged.](#)

Info-09: Event is never emitted

Severity: Informational

Category: Code clarity

File(s): [KickerActions.sol](#)

Bug description:

The following are defined but never emitted. They can be removed to make the code cleaner.

Affected code:

- [src/libraries/external/KickerActions.sol](#)

```
# File: src/libraries/external/KickerActions.sol
```

```
KickerActions.sol:81:      event RemoveQuoteToken(address indexed lender, uint256 indexed price,  
uint256 amount, uint256 lpRedeemed, uint256 lup);
```

```
KickerActions.sol:83:      event BucketBankruptcy(uint256 indexed index, uint256 lpForfeited);
```

Ajna's response:

[Acknowledged.](#)

Info-10: Change uint to uint256

Severity: Informational

Category: Code style

File(s): [PoolInfoUtilsMulticall.sol](#), [Loans.sol](#)

Bug description:

Throughout the code base, some variables are declared as `uint`. To favor explicitness, consider changing all instances of `uint` to `uint256`

Affected code:

- [src/PoolInfoUtilsMulticall.sol](#)

```
# File: src/PoolInfoUtilsMulticall.sol
```

```
PoolInfoUtilsMulticall.sol:294:      for (uint i = 1; i < strBytes.length / 2; ++i) {
```

- [src/libraries/internal/Loans.sol](#)

File: src/libraries/internal/Loans.sol

```
Loans.sol:121:    function _bubbleUp(LoansState storage loans_, Loan memory loan_, uint index_)
private {

Loans.sol:137:    function _bubbleDown(LoansState storage loans_, Loan memory loan_, uint index_)
private {

Loans.sol:139:        uint cIndex = index_ * 2;

Loans.sol:166:    function _insert(LoansState storage loans_, Loan memory loan_, uint index_,
uint256 count_) private {
```

Ajna's response:

[Acknowledged.](#)

Gas-01: State variables only set in the constructor should be declared immutable

Severity: Informational

Category: Gas optimization

File(s): [ERC20PoolFactory.sol](#), [ERC721PoolFactory.sol](#)

Bug description:

Variables only set in the constructor and never edited afterwards should be marked as immutable, as it would avoid the expensive storage-writing operation in the constructor (around 20 000 gas per variable) and replace the expensive storage-reading operations (around 2100 gas per reading) to a less expensive value reading (3 gas)

Affected code:

- [src/ERC20PoolFactory.sol](#)

Unset

File: src/ERC20PoolFactory.sol

```
ERC20PoolFactory.sol:35:        implementation = new ERC20Pool();
```

- [src/ERC721PoolFactory.sol](#)

Unset

```
# File: src/ERC721PoolFactory.sol
```

```
ERC721PoolFactory.sol:37:      implementation = new ERC721Pool();
```

Ajna's response:

Acknowledged.

Gas-02: uint256 to bool mapping: Utilizing Bitmaps to dramatically save on Gas

Severity: Informational

Category: Gas optimization

File(s): [RewardsManager.sol](#), [ERC721Pool.sol](#)

Bug description:

<https://soliditydeveloper.com/bitmaps>

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/BitMaps.sol>

- [BitMaps.sol#L5-L16](#):

Unset

```
/**
```

```
 * @dev Library for managing uint256 to bool mapping in a compact and  
 efficient way, provided the keys are sequential.
```

```
 * Largely inspired by Uniswap's
```

```
 https://github.com/Uniswap/merkle-distributor/blob/master/contracts/MerkleDi  
 stributor.sol[merkle-distributor].
```

```
 *
```

```
 * BitMaps pack 256 booleans across each bit of a single 256-bit slot of  
 `uint256` type.
```

```
 * Hence booleans corresponding to 256 _sequential_ indices would only  
 consume a single slot,
```

```
* unlike the regular `bool` which would consume an entire slot for a single
value.
*
* This results in gas savings in two ways:
*
* - Setting a zero value to non-zero only once every 256 times
* - Accessing the same warm slot for every 256 _sequential_ indices
*/
```

Affected code:

- [src/ERC721Pool.sol](#)

Unset

```
# File: src/ERC721Pool.sol
```

```
ERC721Pool.sol:81:    mapping(uint256 => bool)    internal
tokenIdsAllowed_;
```

- [src/RewardsManager.sol](#)

Unset

```
# File: src/RewardsManager.sol
```

```
RewardsManager.sol:73:    mapping(uint256 => mapping(uint256 => bool))
public override isEpochClaimed;
```

Ajna's response:

[Acknowledged.](#)

Gas-03: Increments/decrements can be unchecked in for-loops

Severity: Informational

Category: Gas optimization

File(s): [PoolInfoUtilsMulticall.sol](#), [SafeTokenNamer.sol](#)

Bug description:

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

The change would be:

```
Unset
- for (uint256 i; i < numIterations; i++) {
+ for (uint256 i; i < numIterations;) {
  // ...
+   unchecked { ++i; }
}
```

These save around 25 gas saved per instance.

The same can be applied with decrements (which should use `break` when `i == 0`).

The risk of overflow is non-existent for `uint256`.

Affected code:

- [src/PoolInfoUtilsMulticall.sol](#)

```
Unset
# File: src/PoolInfoUtilsMulticall.sol

PoolInfoUtilsMulticall.sol:132:      for(uint256 i = 0; i <
functionSignatures_.length; i++) {

PoolInfoUtilsMulticall.sol:198:      for (uint256 i = 0; i < numDelimiters
- 1; i++) {

PoolInfoUtilsMulticall.sol:236:      for (uint256 i = startIndex_; i <=
strBytes.length - patternBytes.length; i++) {

PoolInfoUtilsMulticall.sol:238:      for (uint256 j = 0; j <
patternBytes.length; j++) {
```

```
PoolInfoUtilsMulticall.sol:257:      for (uint256 i = startIndex_; i <=
endIndex_; i++) {

PoolInfoUtilsMulticall.sol:267:      for (uint256 i = 0; i <
strBytes.length; i++) {

PoolInfoUtilsMulticall.sol:294:      for (uint i = 1; i < strBytes.length
/ 2; ++i) {
```

- [src/libraries/helpers/SafeTokenNamer.sol](#)

Unset

```
# File: src/libraries/helpers/SafeTokenNamer.sol
```

```
SafeTokenNamer.sol:60:      for (uint256 j = 0; j < 32; j++) {

SafeTokenNamer.sol:80:      for (uint256 i = 0; i < len / 2; i++) {
```

Ajna's response:

[Acknowledged.](#)

Gas-04: ++i costs less gas compared to i++ or i += 1

Severity: Informational

Category: Gas optimization

File(s): [PoolInfoUtilsMulticall.sol](#), [SafeTokenNamer.sol](#)

Bug description:

Pre-increments are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Note that post-increments return the old value before incrementing, hence the name *post-increment*:

Unset

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after the
comparison");
```

However, pre-increments return the new value:

Unset

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before the
comparison");
```

In the pre-increment case, the compiler has to create a temporary variable (when used) for returning 1 instead of 2.

Saves 5 gas per instance

Affected code:

- [src/PoolInfoUtilsMulticall.sol](#)

Unset

```
# File: src/PoolInfoUtilsMulticall.sol
```

```
PoolInfoUtilsMulticall.sol:132:      for(uint256 i = 0; i <
functionSignatures_.length; i++) {
```

```
PoolInfoUtilsMulticall.sol:198:      for (uint256 i = 0; i < numDelimiters
- 1; i++) {
```

```
PoolInfoUtilsMulticall.sol:212:      start++;
```

```
PoolInfoUtilsMulticall.sol:227:      count_++;
```

```
PoolInfoUtilsMulticall.sol:236:      for (uint256 i = startIndex_; i <=
strBytes.length - patternBytes.length; i++) {

PoolInfoUtilsMulticall.sol:238:      for (uint256 j = 0; j <
patternBytes.length; j++) {

PoolInfoUtilsMulticall.sol:257:      for (uint256 i = startIndex_; i <=
endIndex_; i++) {

PoolInfoUtilsMulticall.sol:267:      for (uint256 i = 0; i <
strBytes.length; i++) {
```

- [src/libraries/helpers/SafeTokenNamer.sol](#)

Unset

```
# File: src/libraries/helpers/SafeTokenNamer.sol
```

```
SafeTokenNamer.sol:60:      for (uint256 j = 0; j < 32; j++) {

SafeTokenNamer.sol:64:      charCount++;

SafeTokenNamer.sol:80:      for (uint256 i = 0; i < len / 2; i++) {
```

Ajna's response:

[Acknowledged.](#)

Formal Verification

Assumptions and Simplifications Made During Verification

General Assumptions

- A. The block timestamp was assumed to be > 0 and $< 2^{96}$.
- B. Any loop was unrolled to two iterations at most.
- C. Recursive function calls (in Loans.sol) are called at most twice (recursive depth of 2).

Deposits library summarization

The implementation of the Deposits.sol library functions was replaced by a summarized implementation that ignores the Fenwick tree structure, and instead makes use of arbitrary mappings (index to value, or sum to index) that correspond to the abstract array modeled by the Fenwick tree. The different functions of the library were assumed to behave according to the following rules:

- A. ***scale()*** returns an arbitrary number.
- B. ***prefixSum()*** returns an arbitrary number.
- C. ***unscaleValueAt()*** returns an arbitrary number.
- D. ***findIndexOfSum()*** is replaced by a mapping(sum \rightarrow index) which is assumed to be weakly monotonically increasing with respect to the index, and is bounded by the MAX_FENWICK_INDEX().
- E. ***unscaledAdd(index)*** is assumed to increase the value returned by *unscaleValueAt(index)* by the unscaled amount, within 1% error.
- F. ***unscaledRemove(index)*** is assumed to reduce the value returned by *unscaleValueAt(index)* by the unscaled amount, within 1% error.
- G. When either of the functions *unscaledAdd(index)* or *unscaledRemove(index)* are called, the index corresponding to the LUP, is changed only if its previous value was larger or equal to the input index. Specifically, when *unscaledAdd()* or *unscaledRemove()* are called, the LUP index is assumed to arbitrarily decrease or increase, respectively.

PoolHelper.sol summarization

Some of the functions in PoolHelper.sol were summarized (but not checked against the real code), such that their behavior was dictated according to the following rules:

- A. **`_priceAt(index)`** was assumed to be strictly monotonic, and bounded between `MIN_PRICE()` and `MAX_PRICE()`. Additionally, we assumed `_priceAt(MAX_BUCKET_INDEX) = WAD()`.
- B. **`_indexOf(price)`** was assumed to be strictly monotonic, and bounded between 0 and `LAST_INDEX()`.
- C. **`_minDebtAmount()`** was assumed to return an arbitrary value.
- D. **`_claimableReserves()`** was assumed to return an arbitrary value.
- E. **`_reserveAuctionPrice()`** assumed to return an arbitrary value.
- F. **`_auctionPrice(reference price, kick time)`** was replaced by the following formula:
Price = 256 * wmul(factor(T - T_kick) , reference price), where:
 - a. “T” is the block time stamp, “T_kick” is the kick time, hence the function only depends on the time difference.
 - b. `factor(x)` is a monotonically decreasing function that satisfies
 - i. `factor(0) = WAD`
 - ii. `factor(1200) = WAD / 2`
 - iii. Values for all other inputs are assumed to be arbitrary, that satisfy the monotonicity property (in particular, *forall* $x \geq 0$, $factor(x) \leq WAD()$).
- G. **`_bpf()`** was assumed to return an arbitrary value (*bpf*) that satisfies the following conditions:
 - a. $abs(bpf) \leq \text{bond factor}$, where ‘abs’ is the absolute value.
 - b. If neutral price == auction price then *bpf* = 0.
 - c. If neutral price > auction price then *bpf* >= 0.
 - d. If neutral price < auction price then *bpf* <= 0.
- H. **`_bondParams(debt, ratio)`** was assumed to return an arbitrary value for the bond factor, depending solely on the ratio. The bond size is calculated by the following formula:
Bond size = wmul(bond factor, debt).

Formal Verification Properties

Notations

✓ Indicates the rule is formally verified.

✗ Indicates the rule is violated.

Since the protocol consists of different contracts, we will present the relative properties for each of the main contracts in separate sections.

The following files were formally verified, and the properties are listed below per library\contract:

- A. KickerActions.sol
- B. Loans.sol
- C. Buckets.sol
- D. LenderActions.sol
- E. TakerActions.sol
- F. BorrowerActions.sol

The functions in Pool.sol and ERC20.sol that aren't involving the external\internal libraries were not formally verified, so that only direct calls to the library functions at an arbitrary state were considered. Furthermore, we didn't formally check any functionality regarding the ERC721 pool.

KickerActions.sol

Assumptions

- We verified the library functions against an arbitrary storage state.
- The pool state inflator doesn't change at all in any function call.

Properties

1. ✓ Lowest Utilized Price (LUP) must be greater or equal to Highest Threshold Price (HTP).
2. ✓ A borrower cannot turn from over-collateralized to under-collateralized.
3. ✓ For any borrower, the kick time is always in the past (or zero if not in auction).

4. ☒ The amount locked for any kicker always covers the bond size for the loan that was kicked.
5. ☒ The change in a kicker locked bond amount plus the claimable amount is equal to the change of total escrowed.
6. ☒ If the kick time is zero, then either the borrower's debt is zero, or the borrower is in the loans array.
7. ☒ A non-lender (zero LPs) cannot kick an overcollateralized borrower (also for lender kick).
8. ☒ If an action changes the borrower kicker from zero to non-zero, then the previous bond size must have been zero, and the msg.sender is registered as the kicker.
9. ☒ The liquidation kicker can only change from the zero address to non-zero address or the opposite.
10. ☒ Kicking borrowerA doesn't affect the data of borrowerB.
11. ☒ The bond size and the liquidation kicker can only be changed for a specific kicker and borrower at a time.
12. ☒ If kicking a borrower which is not the HTP borrower, then the HTP stays the same.
13. ☒ Lender kick cannot succeed for a collateralized borrower and if that lender's deposit time is earlier than the borrower bankruptcy time.
14. ☒ For any method called, if the borrower was not kicked, it has no action going on.
15. ☒ Consistency of any borrower index and loans array is maintained.
16. ☒ The integrity of auctions linked chain is preserved.
17. ☒ A borrower kick time is zero if and only if it is not in the auctions chain.
18. ☒ If a borrower's kick time is non-zero then it doesn't appear in the loans array.

Loans.sol

Assumptions

- We verified the library functions against an arbitrary storage state.
- We assumed one cannot update or remove the zero borrower.

Properties




1. ☒ Consistency of any borrower index and loans array is maintained.
2. ☒ The threshold price at a given *index* is always larger than or equal to the threshold price at the child index ($2 * index$, $2 * index + 1$).
3. ☒ Removing or updating a borrower doesn't affect other borrowers.
4. ☒ Integrity of the borrower data after update() - the data is updated according to the input.

Buckets.sol

Assumptions

- We verified the library functions against an arbitrary storage state.
- We assumed an arbitrary value for each bucket deposit.

Properties







1.  *addCollateral(index, lender, amount)* satisfies:
 - a. If the lender deposit time is larger than the bucket bankruptcy time, then the lender lps is incremented by the returned LPS delta.
 - b. The bucket LPS is incremented by the returned LPS delta.
 - c. The returned LPS delta is equal (up to one unit) to the value of *collateralToLP(amount)*.
 - d. If the added collateral is zero, then the LP amount minted is zero.
 - e. The LPS of any other lender is not changed.
 - f. The LPS of any other bucket is not changed.
2.  LP to quote tokens round trip equality: ¹
lpToQuoteTokens(quoteTokensToLP(amount)) == amount
3.  LP to quote tokens round trip cannot yield any gain: ²
lpToQuoteTokens(quoteTokensToLP(amount)) <= amount

LenderActions.sol

Assumptions

- We verified the library functions against an arbitrary storage state

Properties

1.  Invariant: For any method called, the Lowest Utilized Price (LUP) must be greater or equal to the Highest Threshold Price (HTP). (*Lup_GE_HTP*) ³
2.  Invariant: For any method called, any borrower index and loans array is maintained. (*addressConsistency*)
3.  Invariant: For any method called, always the max threshold price is stored at the first index. (*MaxThresholdPriceIsAtFirstIndex*)
4.  Invariant: For any method called, if the borrower was not kicked, it has no action going on. (*NotInAuctionNotLinked*)
5.  Invariant: For any method called, the integrity of auctions linked chain is preserved. (*AuctionsChainIntegrity*)
6.  For *addCollateral()* the added amount cannot be zero (*addCollateralAmountNotZero*)

¹ A more lenient condition that the difference between amount and the round trip value is less than 10^9 is also violated.

² Both calculations use rounding down.

³ Could not be verified for the *moveQuoteToken()* method due to over-approximation of the LUP index calculation



7.  For `addCollateral()` the returned bucketLP amount cannot be zero
(addCollateralBucketLPNotZero)
8.  For `addCollateral()` the used index must be in the legal range: $0 < \text{index} \leq \text{MAX_FENWICK_INDEX} = 7_388$ (addCollateralLegalIndex)
9.  For `addCollateral()` the more collateral one adds to a bucket, the returned bucket LP is higher
(addCollateralMonotonicityOfLP)
10.  For `addQuoteToken()` the added amount cannot be zero (addQuoteTokenAmountNotZero)
11.  For `addQuoteToken()` the returned bucketLP amount cannot be zero
(addQuoteTokenBucketLPNotZero)
12.  For `addQuoteToken()` the returned LUP is updated correctly
(addQuoteTokenCorrectLUPUpdate)
13.  For `addQuoteToken()` the used index must be in the legal range: $0 < \text{index} \leq \text{MAX_FENWICK_INDEX} = 7_388$ (addQuoteTokenLegalIndex)
14.  For `moveQuoteToken()` the *from* and *to* buckets must be different
(moveQuoteTokenFromAndToBucketsAreDifferent)
15.  For `moveQuoteToken()` only the *from* and *to* buckets are affected
(moveQuoteTokenOnlyAffectsTheSpecificIndexBucket)
16.  When user calls `moveQuoteToken()` he cannot affect the LP of any of the other lenders
(moveQuoteTokenOtherLenderLPStaysTheSame)
17.  For `removeCollateral()` the added amount cannot be zero
(removeCollateralMustNotBeZero)
18.  `removeCollateral()` affects only one specific bucket
(removeCollateralOnlyAffectsTheSpecificIndexBucket)
19.  Calling `removeQuoteToken()` cannot increase the Lowest Utilized Price (LUP)
(removeQuoteTokenCannotIncreaseLUP)
20.  `removeQuoteToken()` affects only one specific bucket
(removeQuoteTokenOnlyAffectsTheSpecificIndexBucket)
21.  After calling `removeQuoteToken()` the removed amount cannot be zero
(removeQuoteTokenRemovedAmountMustBeNonZero)
22.  When user calls `removeQuoteToken()` he cannot affect the LP of any of the other lenders
(removeQuoteTokenOtherLenderLPStaysTheSame)

TakerActions.sol

Assumptions

- We verified the library functions against an arbitrary storage state.
- Some rules were verified by assuming a value of $1.01 \cdot \text{WAD}$ (1%) for the inflator.

Properties

1.  Lowest Utilized Price (LUP) must be greater or equal to Highest Threshold Price (HTP). [L-03](#)
2.  A borrower cannot turn from over-collateralized to under-collateralized. [L-03](#)
3.  The change in a kicker locked bond amount plus the claimable amount is equal to the change of total escrowed.
4.  *take()* doesn't change any bucket's collateral.
5.  For *bucketTake()*, the bucket price is always larger or equal than the current auction price.
6.  After *bucketTake()*, the bucket's collateral is changed correctly, and doesn't change any other bucket.
7.  For any kind of take, the collateral amount taken is never larger than the borrower's current collateral.
8.  Any kind of take doesn't change any other auction variables for another borrower.
9.  If an auction is settled, all of its variables turn to zero, and the number of auctions is reduced by one.
10.  If an auction is not settled, then its kicker, kick time, reference price, neutral price, its next and previous auctions on the list, and the number of auctions don't change.
11.  For any kind of take:
 - a. The auction is settled if and only if the borrower's debt turns to zero.
 - b. The auction is settled if and only if the number of auctions is reduced by one.
 - c. The auction is settled if and only if the kick time is set back to zero.
 - d. The take result remaining collateral equals the result collateral post action.
 - e. The take result collateral pre auction is the actual collateral of the borrower.
12.  For any borrower, the kick time is always in the past (or zero if not in auction).
13.  If the kick time is zero, then either the borrower's debt is zero, or the borrower is in the loans array.
14.  The amount locked for any kicker always covers the bond size for the loan that was kicked.
15.  The pool *t0_debt* is larger or equal than any single borrower *t0_debt*.
16.  Integrity of *calculateTakeFlowsAndBondChangeIntegrity()* : The repay amount *t0* is never larger than the borrower *t0* debt. [L-01](#)
17.  The integrity of auctions linked chain is preserved.

18. ☒ A borrower kick time is zero if and only if it is not in the auctions chain.
19. ☒ If a borrower's kick time is non-zero then it doesn't appear in the loans array.

BorrowerActions.sol

Assumptions

- We verified the library functions against an arbitrary storage state.
- `PoolState.inflator` is greater or equal to `WAD`.

Properties

1. ☒ `drawDebt()` integrity: collateral balances of borrower and pool are updated correctly (`drawDebtIntegrity`)
2. ☒ `repayDebt()` integrity: collateral balances of borrower and pool are updated correctly (`repayDebtIntegrity`)
3. ☒ `drawDebt()` doesn't change other users' debts (`drawDebtCantHarmOthers`)
 - a. `inflator` is greater or equal to `WAD`
4. ☒ `repayDebt()` doesn't change other users' debts (`repayDebtCantHarmOthers`)
 - a. `inflator` is greater or equal to `WAD`
5. ☒ `drawDebt()` preserves correlation $poolDebt == t0PoolDebt * inflator$ (`debtAndDebt0Consistency_drawDebt`)
 - a. `inflator` is between `WAD` and `WAD*4`
6. ☒ `repayDebt()` preserves correlation $poolDebt == t0PoolDebt * inflator$ (`debtAndDebt0Consistency_repayDebt`)
 - a. `inflator` is between `WAD` and `WAD*4`
7. ☒ No collateral losses for the pool after the user's round-trip (`drawDebt`, then `repayDebt` called at the same block) starting from 0 user's collateral (`t(noRoundTripBenefits)`)
8. ☒ A user can't have a debt without collateral (`noCollateralNoDebt`)
 - a. `inflator` is between `WAD` and `WAD*4`
9. ☒ `t0debt` of address 0 is always 0.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.

