PUBLIC

Code Assessment

of the Xena Smart Contracts

October 3, 2023

Produced for

Xena FINANCE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Xena Finance team,

Thank you for trusting us to help Xena Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Xena according to Scope to support you in forming an opinion on their security risks.

Xena Finance implements a decentralized, non-custodial perpetual exchange. It aims to provide users with zero price-impact trades.

The most critical subjects covered in our review are asset solvency and functional correctness. Security regarding the aforementioned subjects is improvable. The most important issues uncovered are:

- Asset solvency is low due to wrongly maintained internal accounting, see Wrong Accounting upon Margin Account Top up.
- Functional correctness is low due to the value the tranches not including unrealized LP fees, see Accrued Interest Is Not Accounted in trancheValue.

The first issue has been fixed by a change of specification. Xena Finance has decided they only want to use a single tranche. The issue remains valid if Xena Finance decides to add more tranches. This leaves the codebase complex, while the functionality that will be used is simpler. The second issue related to accrued interest remains unfixed.

Additionally, there are a number of issues that Xena Finance decided not to fix, which could cause problems in the edge cases outlined in those issues.

The general subjects covered are documentation and specification. Security regarding all the aforementioned subjects is improvable. Documentation and specification are not sufficient due to the overall lack of documentation and unclear specification, see Missing Documentation.

In summary, we find that the codebase currently provides an improvable level of security.

Users of the system should check the Notes section for important information to consider before using the system.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Specification Changed	1
Medium-Severity Findings	6
• Code Corrected	2
• Risk Accepted	3
• Acknowledged	1
Low-Severity Findings	12
• Code Corrected	4
• Specification Changed	1
• Code Partially Corrected	1
• Risk Accepted	4
• Acknowledged	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Xena repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	21 Aug 2023	be87d8a5c0dbc03fa801e69b0b3a87bb863cb43c	Initial Version
2	28 Aug 2023	e2958010572483509bf3c6ae10f48c17f7a1425c	Updated Version
3	2 Oct 2023	e2ac43efb81687360d8b36f1bda637ee1d84f281	Version 3 with fixes

For the solidity smart contracts, the compiler version `0.8.18` was chosen.

The following contracts are in the scope of the review:

```
interest:
    SimpleInterestRateModel.sol

interfaces:
    AggregatorV3Interface.sol
    ILPToken.sol
    IOracle.sol
    IOrderManagerWithStorage.sol
    IPoolWithStorage.sol
    ITradingIncentiveController.sol
    IETHUnwrapper.sol
    ILiquidityCalculator.sol
    IOrderHook.sol
    IPool.sol
    IReferralController.sol
    IWETH.sol
    IInterestRateModel.sol
    IMintableErc20.sol
    IOrderManager.sol
    IPoolHook.sol
    ITradingContest.sol

lens:
    OrderLens.sol
    PoolLens.sol

lib:
    Constants.sol
    DataTypes.sol
    MathUtils.sol
    PositionLogic.sol
    SafeCast.sol
```

```
    SafeERC20.sol
    SignedIntMath.sol

oracle:
    Oracle.sol
    PriceReporter.sol

orders:
    OrderManager.sol
    OrderManagerStorage.sol

pool:
    LiquidityCalculator.sol
    Pool.sol
    PoolStorage.sol

tokens:
    LPToken.sol

utils:
    ETHUnwrapper.sol
    LiquidityRouter.sol
    OrderHook.sol
    PoolHook.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly listed above are out of the scope of this review. Third-party libraries (e.g., `FixedPointMathLib`, `SafeTransferLib`, etc.) are out of the scope of this review.

`OrderHook` and `PoolHook` make calls to `ITradingIncentiveController`, `IReferralController` and `ITradingContest`, for which no implementation is given. As a result, any calls to these contracts are assumed to be correct.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Xena Finance offers Xena, a decentralized, non-custodial perpetual exchange. It aims to provide users with zero price-impact trades.

The system allows to open long positions where the index and the collateral tokens are the same and cannot be stablecoins, short positions where the index token is not a stablecoin and the collateral token is a stablecoin, and swaps between the assets in the pool. Stablecoins in Xena are USD-pegged stablecoins. The main components are the `Pool`, the `OrderManager`, and the `Oracle`.

2.2.1 Pool

The Pool holds the Liquidity providers' assets. These assets can be used to trade against, where the LPs always take exact opposite trade to the traders. If the trader goes long on an asset, the LPs go short. Trades can either be normal swaps, or leveraged positions requiring collateral. The Pool also holds the collateral of users that have a leveraged position, as well as any accrued DAO fees that have not been claimed.

There is only one Pool, which can hold many different tokens. Each token has a target weight, which is set by the pool controller. Prices of tokens depend solely on an external oracle. Unlike AMM-style DEXes, the ratio of tokens in the pool does not determine the price. This means that there is no price-impact from trades, no matter their size. The fees charged for a trade, however, are dependent on the token ratios in the pool. A trade that brings the ratio closer to the target weight will receive a fee discount, while a trade that moves the pool away from the target weight will be charged additional fees.

Tranches:

The Pool is split into different "tranches". According to the documentation, these tranches should allow LPs to choose their desired level of risk vs reward. The tranches are differentiated by having a different `riskFactor`, which is set per token. The `riskFactor` determines the proportions with which the different tranches participate in fulfilling swaps and leverage orders. For example, assume there are two tranches A and B, with risk factors for `ETH` set to `10` and `5`. When a swap occurs, `2/3` of the `tokenIn` will go to tranche A and `2/3` of the `tokenOut` will be taken from tranche A. The rest of the tokens will be swapped through tranche B. The fees generated from the swap will also be attributed to the tranches in the same proportions. If, however, tranche A did not have enough `tokenOut` to cover `2/3` of the `swapAmount`, then the maximum number of tokens available will be used from tranche A, and the rest will be used from tranche B. In this case, more than `1/3` of the swap would be fulfilled by tranche B, leading to a utilization that is higher than its `riskFactor`. However, tranche A will still receive `2/3` of the fees, according to its `riskFactor`. The reduced risk from tranche B being a "more senior" tranche (lower `riskFactor`) only comes from reduced utilization. The `risk/utilization` ratio is equal to the other tranches in the normal case. However, if the other tranches are already fully utilized, the more senior tranches can also be fully utilized, which leads to full risk exposure, without receiving any additional fees. Full utilization may be most likely in scenarios with high volatility. This can cause large losses to LPs due to price movements, which could be inflicted to senior and junior tranches. In some cases, the senior tranche loss percentage could even outweigh that of junior tranches.

User functions:

The following functions can be called by users:

- `addLiquidity`: LPs can add liquidity to a tranche for one token listed in the pool at a time. An entry fee is taken when adding liquidity. It behaves in the same way as the dynamic trading fee described above, rewarding the addition of assets for which the pool is below the target weight. The amount of LP tokens is computed from the USD value added after fees and follows the formula `valueAddedUSD / trancheValueUSD * totalSupplyTrancheLPTokens`. LPs can add liquidity in a token up until the token's max liquidity in the pool is reached. There is one LP token per tranche, they are burnable and transferable. LPs can set slippage protection. The pool value is recomputed upon liquidity provision.
- `removeLiquidity`: LPs can remove their liquidity in any of the asset tokens, no matter which token they provided liquidity in. An exit fee is taken when removing liquidity, it behaves in the same way as the dynamic trading fee described above. The specified amount of LP tokens will be burned and the corresponding USD value in the exiting token will be sent back to the LP. LPs can set slippage protection. The pool value is recomputed upon liquidity deprovision.
- `swap`: users can swap any listed token for any asset token without price impact, at the most recent price reported by the Oracle. Dynamic trading fees are computed on both tokens and the highest one will be applied. For non-stable tokens, there is a minimum fee of 0.1% on all swaps. The fee is taken in input token. Swaps can happen as long as there is enough unreserved liquidity in the pool and the input token's max liquidity in the pool is not reached. After the swap, the `postSwap` function is called on the `poolHook` if the address is configured.

- `liquidatePosition`: allows anyone to liquidate an unhealthy leveraged position. A position is liquidatable if one of the following conditions is met:

    1. The collateral value cannot cover the trader's loss.
    2. The `(collateralValue + unrealizedPnL)` is smaller than the maintenance margin, which is a percentage of a position's size.
    3. The `(collateralValue + unrealizedPnL)` is smaller than the sum of the due fees and liquidation fee.

Note that `unrealizedPnL` is negative if the trader is making a loss. If a position is liquidated without loss for the pool, the position is closed (exit fee is taken), the collateral value minus the trader's loss is sent back to the position's owner, and the liquidator receives a reward. If the collateral of the position is not enough to cover the loss, the accounting is updated such that the excess loss is taken by LPs. The trader does not receive back any collateral, and the liquidator receives a reward.

Note that by default, the fee distribution is 100% to the DAO after the pool initialization. This can be changed later by the pool `owner`.

2.2.2 OrderManager

The `OrderManager` is responsible for managing leveraged position updates and swap orders. Both are placed by traders, but only a special whitelisted address, the `executor`, can execute them in the pool, unless "public execution" is enabled. If it is enabled, the owner of the order can execute it as well. Note that swaps can be executed directly through the `OrderManager` without creating an order, which is equivalent to a "market order" order type, or by placing a swap order, which is equivalent to a "limit order". The `OrderManager` will hold all the funds related to pending orders and execution fees.

Swaps:

The following functions are available to users for managing swaps:

- `swap`: executes a swap on the pool at the current market price. If the `orderHook` address is configured, `preSwap` is called at the start of the function. For `swap`, the Pool assumes that tokens are sent to the Pool beforehand. It does not `transferFrom` the caller. The `swap` function of `OrderManager` helps users first transfer tokens, then call `swap` in the Pool, within the same transaction.
- `placeSwapOrder`: creates a swap order that can be executed later by the `executor`, or the order's owner if public execution is allowed. Users can specify the input/output tokens, the amount in, the minimum amount out for slippage protection, a price that is never used, and some extra data for the `orderHook`. When a new order is created, the amount of `tokenIn` is transferred from the caller to the `OrderManager`, and an execution fee is retained in ETH. Then it is assigned a unique id, and `orderHook.postPlaceSwapOrder` is called if the `orderHook` address was configured. When executed, the `OrderManager` will call `Pool.swap` and send the execution fee to an address provided by the caller. If the `tokenOut` was ETH, the first recipient is the `OrderManager`, that will unwrap the received `WETH` and send it to the order's owner. By default, the status of an order is `OPEN`, once executed it is `FILLED`. Only `OPEN` orders with an owner can be executed.
- `cancelSwapOrder`: the owner of an order can cancel it. The token amount as well as the execution fee will be returned to the order's owner and the order's status is set to `CANCELLED`. Only `OPEN` orders can be cancelled.

Leverage orders:

Leverage orders in Xena are "synthetic". No assets are bought or sold when opening or closing a leverage order. Instead, the position `size` of assets is reserved from the LP pool. This ensures that no matter the future price, there will always be enough reserved assets to pay the trader the payoff of their long or short. For longs, an amount of index token is reserved. This means that even if the price went to infinity, the system would have enough to pay the trader. For shorts, an amount of stablecoin is reserved.

This means that even if the index token's price went to zero, the system would be able to pay the trader for the maximum payoff of a short (+100%). It is not possible to open long or short positions with a stablecoin as index token.

The following functions are available to users for managing leverage orders:

- `placeLeverageOrder`: creates a leverage order that can be executed later by the `executor`, or the order's owner if public execution is allowed. Users can specify whether they want to `INCREASE` or `DECREASE` the position's size or collateral, the type of position (`LONG` or `SHORT`), the index and collateral tokens, whether the order is a market order or not, and some data. Upon increasing a position, the data encodes the limit price for the order, which token the trader wants to pay (`payToken`), the amount of collateral, the size of the increase in USD, and some extra data for the `orderHook`. When a new order is created, the amount of `payToken` is transferred from the caller to the `OrderManager`, and an execution fee is retained in ETH. Then it is assigned a unique id, and `orderHook.postPlaceOrder` is called if the `orderHook` address was configured. When executed, the `OrderManager` will first check the validity of the order, as well as the expiration timestamp and compare the order's price with the market price. If the order has expired, the token amount and execution fee are returned to the order's owner. Otherwise, it will swap the `payToken` into the `collateralToken` if they are different. Depending on the position update type, `Pool.increasePosition()` or `Pool.decreasePosition()` will be called:

    - `Pool.increasePosition()`: interest is accrued on the collateral token and the position's fee is computed. The position fee consists of a fee based on the size of the change, plus the dynamic fee described above. Then, the asset will be reserved. If the position size is increased, the collateral and reserved amount will be distributed according to the index token's risk factor of each tranche. For shorts, the full available liquidity can be reserved. For longs, a certain amount always stays unreserved, in order to allow LP withdrawals. The DAO fee is deducted, then the LP fee is distributed according to the risk factor of each tranche. The reserved amount is updated according to the distribution computed above. For a long position, the collateral is added to the `poolAmount` of the tranche, and the guaranteed value is increased by the reserved amount that belongs to the LP, plus fees. In the case of a short position, the new short is added to the global short position, and it is enforced to be within the maximum size for the token. At the end, a check is done to ensure the position does not use more than the maximum allowed leverage and that it is not immediately liquidatable. It is also not allowed to have a position with less than `1x` leverage. The `poolHook.postIncreasePosition` function will be called at the end of the execution if the `poolHook` address is configured.

    - `Pool.decreasePosition()`: interest is accrued on the collateral token and the position's fee is computed. The payout is computed, and the fees are taken from the PnL in case of a gain from the trader, or from the remaining collateral in the case of a loss. The PnL and fees are taken proportional to the position' size change. Then the assets are released. The DAO fee is added in a special mapping, then the LP fee is distributed according to the risk factor of each tranche. The reserved amount is released in the same proportions it has been recorded to be reserved across the tranches. The pool amount is reduced or increased depending on whether the trader made a gain or a loss, according to the distribution computed above. If the position is a long, the guaranteed value is decreased and if it is a short, the global short position is decreased. At the end, a check is done to ensure the position does not use more than the maximum allowed leverage and that it is not liquidatable. Also, it is not allowed to overcollateralize a position. The `poolHook.postIncreasePosition` function will be called at the end of the execution if the `poolHook` address is configured.

By default, the status of an order is `OPEN`, once executed it is `FILLED`. Only `OPEN` orders with an owner can be executed.

- `cancelLeverageOrder`: the owner of an order can cancel it. The token amount as well as the execution fee will be returned to the order's owner and the order's status is set to `CANCELLED`. Only `OPEN` orders can be cancelled.

Note that due to the way that leverage orders are implemented (synthetic leverage through LP asset reservation), it is not relevant for the overall pool value whether or not a position is liquidated in time. Liquidations merely update the internal accounting.

Note that interest is accrued for a token every time its utilization in the pool changes, i.e., upon add/remove liquidity, swaps, and position management. The interest is non-compounding and is accrued per periods of 1 hour.

2.2.3 Oracle

The `Oracle` is used to get the `USD` price of tokens. The primary source of prices is the `reporter` role, which is assumed to be held by the `PriceReporter` contract, controlled by a Keeper. The Keeper posts prices, which are then compared with a reference price (Chainlink). If the Keeper's price is within `chainlinkDeviation` of the Chainlink price and has been updated within the last 5 minutes, then the Keeper's price is used. The `chainlinkDeviation` is configured separately per token. If the Keeper's price is not within `chainlinkDeviation`, then the Keeper or Chainlink price will be used, whichever is more favorable for the protocol (for example, `tokenIn` will be undervalued and `tokenOut` will be overvalued). At most, the price used can be `3 * chainlinkDeviation` away from the Chainlink price. If the Keeper's last price is more than 5 minutes old, the Chainlink price will be used with a `0.2%` extra margin in favor of the protocol. If the Keeper's last price is more than 1 hour old, the Chainlink price with an extra 5% will be used. Orders can only be executed if the last Keeper update is more recent than the order creation time.

2.2.4 Trust Model

Users: not trusted

Pool's owner: trusted to correctly set/manage the different parameters and addresses of the pool, as well as the tokens that are listed in the pool and the different tranches and their parameters in a non-adversarial manner. Can pause and unpause the Pool. Note that while paused, LPs cannot withdraw liquidity and open positions cannot be closed. For considerations on tokens and parameters, see Market Manipulation Of Listed Assets Must Not Be Profitable.

Order manager's owner: trusted to correctly set/manage the different parameters and addresses of the `OrderManager` in a non-adversarial manner.

Order manager's controller: trusted to correctly set/manage the minimum execution delay time and public execution flag of the `OrderManager` in a non-adversarial manner.

Executor: See Transaction Ordering MEV. Assumed to be the `PriceReporter` contract.

Liquidity calculator's owner: trusted to correctly set/manage the different fee parameters of the `LiquidityCalculator` in a non-adversarial manner.

Oracle's owner: trusted to correctly set/manage the different price reporters in a non-adversarial manner.

Oracle's whitelisted reporters: trusted to provide correct prices and have a very high uptime. Price reporter's owner: trusted to correctly set/manage the different price reporters in a non-adversarial manner.

Price reporter's whitelisted reporters: trusted to provide correct prices, and to not censor or unfairly reorder orders.

Pool hook's owner: trusted to correctly set/manage the trading contest and trading incentive controller of the `PoolHook` in a non-adversarial manner.

For any contracts that are deployed using proxies: The proxy's owner is fully trusted and could take control of any tokens the proxy controls at any time, including all user funds.

Tokens: any tokens with non-standard ERC20 behaviors (e.g., rebasing, with fee on transfer, reentrant (e.g., ERC-777)) should not be used in Xena. Tokens with 0 decimals and tokens with a large number of decimals (close to 30 or more), should not be used in Xena.

2.2.5 Changes in V3

- In Version 3 it was specified that the system is intended to be used with only one tranche, not multiple as was initially intended.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	4

- Accrued Interest Is Not Accounted in trancheValue Risk Accepted
- Hardcoded Stablecoin Price Risk Accepted
- LP Fee Distribution Is Unfair Acknowledged
- Missing Documentation Risk Accepted

Low-Severity Findings	7

- Contracts Not Implementing Their Interface Risk Accepted
- Free Leverage Within Accrual Interval Risk Accepted
- Incorrect Fee Calculation When Oracles Disagree Risk Accepted
- Missing Reentrancy Protection Code Partially Corrected
- No Slippage Protection on poolSwap Risk Accepted
- OrderLens Marks Inexistent Swap Orders as Executable Acknowledged
- OrderLens Missing Checks for Executable Leverage Orders Acknowledged

5.1 Accrued Interest Is Not Accounted in `trancheValue`

Design Medium Version 1 Risk Accepted

CS-XENA-002

The interest that is owed to LPs by an open leveraged position is only accounted for when that position is updated (increased or reduced), in `_calcPositionFee()`.

If a position is opened, but then no longer updated for a long time, a significant amount of interest may accrue. This pending interest will not be calculated in `_getTrancheValue()`, leading to an undervaluation of LP shares.

Consider the following situation:

There are 2 LPs, both with an equal amount of liquidity. A trader opens a position. The position is open for 1 year and is paying 50% APR in interest. After a year, one of the LPs leaves. One minute later, the trader closes their position. Now, the trader will pay the full interest amount to the remaining LP, even though the risk of the position was shared equally among both LPs.

A third LP could even front-run the transaction which closes the position, depositing an equal amount as the remaining LP to the pool. The trader will now pay half of their accrued interest to the third LP, even though they did not take any risk. The third LP could immediately withdraw their liquidity afterward, receiving a risk-free profit.

The effect of this will be larger, the longer positions remain open without being updated. If positions typically do not stay open for a long time, the accrued interest will likely be small enough that the undervaluation of LP shares is negligible.

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a code change.

5.2 Hardcoded Stablecoin Price

Correctness | Medium | Version 1 | Risk Accepted

CS-XENA-004

In `_getCollateralPrice()`, if the collateral asset is marked as a stablecoin, the value is hardcoded to `1 USD`. The configured oracle is not queried.

In case a stablecoin loses peg, this price will not match the oracle price.

The PnL (against `USD`) of shorts is always paid out as if the stablecoin was worth `1 USD`, no matter the actual value. The collateral is valued consistently between increasing and decreasing a position.

However, in `_getTrancheValue()` of `LiquidityCalculator`, the stablecoins are valued at their oracle price. The PnL of shorts is calculated in USD, independently of the current stablecoin price.

Consider the following illustrative situation:

1. A pool has one tranche and no opening and trading fee, with `2000 USDC` liquidity. A trader has an open short position with `100 USDC` collateral. They currently have a positive PnL of `100 USD`. The tranche has reserved `1000 USDC` of `collateralToken` from LPs. The oracle price of `USDC` is `1 USD`. The `trancheValue` will be calculated as `(2000 - 1000) * 1 - 100 = 900`.
2. The oracle price of `USDC` collapses to zero.
3. Now, the `trancheValue` will be calculated as `(2000 - 1000) * 0 - 100 = -100`.
4. The trader closes their short. They will be paid out their collateral and `100 USDC` (worth `0 USD`).
5. Now, the `trancheValue` will be calculated as `(1900 * 0) = 0`.

In this extreme (and unlikely) example, the system invariant that `AUM` (`trancheValue`) must always be positive, can be broken. This would lead to `_getTrancheValue` always reverting when casting `toUint256()`, which will make it impossible to add or remove liquidity from that tranche.

```
// aum MUST not be negative. If it is, please debug
    return aum.toUint256();
```

A price collapse of the stablecoin to zero is the most extreme case, but the same effect on `trancheValue` happens in a smaller way as soon as the oracle price of the stablecoin is not exactly `1 USD`.

The incorrect `trancheValue` will lead to LP shares being over- or undervalued, which can lead to losses for LPs.

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a code change.

5.3 LP Fee Distribution Is Unfair

Design | Medium | Version 1 | Acknowledged

CS-XENA-018

Upon position increase/decrease/liquidation, the LP fee is scaled and distributed according to the risk factor, and not according to the distribution of the reserve amount across the tranches. In the case where the system has three tranches (junior, mezzanine, and senior) and the junior tranche is full, the risk for newly opened positions will only be distributed across the mezzanine and senior tranches. But in this case, the junior tranche will still receive a share of the LP fee, although it does not participate in the risk, and the mezzanine and senior tranches will not get rewarded according to the new risk.

Consider the following situation:

1. There is a pool with two tranches. Tranche A has `2/3` of the total `riskFactor`, tranche B has `1/3`. Tranche A has only one LP, Alice.
2. Each time a trader wants to take leverage, Alice front-runs the `increasePosition` call of the `executor` with `removeLiquidity`, removing her entire balance.
3. When `increasePosition` is executed, there is no unreserved liquidity in tranche A, so the full amount is reserved in tranche B.
4. Alice deposits her balance again.
5. When the trader closes their leveraged position, Alice receives `2/3` of the `positionFee`, as well as the `borrowFee` (interest), even though she did not provide any leverage to the trader.

Upon a swap, the LP fee is similarly scaled and distributed according to the risk factor, and not according to the distribution of the `amountOut`. So, the LP fee does not reflect the liquidity utilization.

Acknowledged:

Xena Finance acknowledged the issue with the following response:

```
This is intended behavior
```

5.4 Missing Documentation

Design | Medium | Version 1 | Risk Accepted

CS-XENA-005

The codebase is poorly documented and almost no natspec has been written.

Xena Finance did not supply any code-external documentation. Only a reference to a third-party project's documentation website with similar functionality was given.

A well-documented codebase helps integrators and improves the overall security by allowing readers to better understand the role of a piece of code, as well as any assumptions that are made.

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a change.

5.5 Contracts Not Implementing Their Interface

Design Low Version 1 Risk Accepted

CS-XENA-008

Some contracts do not implement their interface, which could lead to problems when integrated.

- `Pool` should implement `IPoolWithStorage`
- `LPToken` should implement `ILPToken`
- `Oracle` should implement `IPriceFeed`
- `OrderManager` should implement `IOrderManagerWithStorage`

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a code change.

5.6 Free Leverage Within Accrual Interval

Design Low Version 1 Risk Accepted

CS-XENA-009

The interest for leveraged positions accrues once per `accrualInterval`.

As a result, a trader could avoid paying any interest by creating a leveraged position after interest has been accrued, then closing the position again before the next accrual. The `positionFee` will still be paid.

Risk accepted:

Xena Finance understands and acknowledges this issue, but has decided not to make a code change.

5.7 Incorrect Fee Calculation When Oracles Disagree

Correctness Low Version 1 Risk Accepted

CS-XENA-011

In `LiquidityCalculator`, `_calcFeeRate()` calculates the fees for a swap based on if the swap moves the pool towards the `targetWeight` or away from it. For this, the value of the token is calculated based on `tokenPrice`. The `targetWeight` is calculated based on the `Pool.virtualPoolValue`.

In the normal case, these values are correct. However, in special conditions, the `Oracle` does not return the Keeper's posted price, but instead gives a price that is more favorable to the protocol. For example, the `tokenPrice` of `tokenIn` for a swap will be undervalued.

The `Pool.virtualPoolValue` is calculated as an average of undervaluing and overvaluing all tokens. This means that in `virtualPoolValue`, the `tokenIn` will not be undervalued in the same way.

```
MathUtils.average(liquidityCalculator.getPoolValue(true), liquidityCalculator.getPoolValue(false));
```

As a result, the weights are computed incorrectly, as values that have different "rounding" applied to them are compared as if they were rounded the same.

Ultimately, this will lead to the fee calculated by `calcFeeRate()` to be either too high or too low.

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a code change.

5.8 Missing Reentrancy Protection

Design | Low | Version 1 | Code Partially Corrected

CS-XENA-014

Although no attack vector for reentrancy or read-only reentrancy was found, the functions `Pool.liquidatePosition` and view function `virtualPoolValue` should implement reentrancy protection to avoid any potential issue.

Code partially corrected:

A reentrancy guard has been added to `Pool.liquidatePosition`.

`virtualPoolValue()` has not been changed, so integrators must be careful when calling this function.

5.9 No Slippage Protection on `poolSwap`

Design | Low | Version 1 | Risk Accepted

CS-XENA-015

The `_poolSwap` function in `OrderManager` has a `_minAmountOut` argument, which can be used for slippage protection.

However, when `_poolSwap()` is called from `_executeLeveragePositionRequest()`, this functionality is not used, always passing a `_minAmountOut` of `0`.

Risk accepted:

Xena Finance understands and accepts the risk posed by this issue, but has decided not to make a code change.

5.10 `OrderLens` Marks Inexistent Swap Orders as Executable

Correctness | Low | Version 1 | Acknowledged

CS-XENA-019

The default status of an order is `OPEN` and the values of an uninitialized order's `amountIn` and `minAmountOut` will be `0`. So, the function `OrderLens.canExecuteSwapOrders` will mark non-existent orders as non-rejected, but they will fail if submitted to the `OrderManager`.

Acknowledged:

Xena Finance answered:

```
We are aware of this.
Contracts don't use this function so we keep that for convenience.
```

5.11 `OrderLens` Missing Checks for Executable Leverage Orders

Design | Low | Version 1 | Acknowledged

CS-XENA-017

The function `OrderLens.canExecuteLeverageOrders` does not do any check for `INCREASE` requests and only does minimal checks for `DECREASE` requests, e.g., the fee is not fully computed. Therefore, it may happen that an order marked as executable by the function will fail if submitted to the `OrderManager`.

Acknowledged:

Xena Finance acknowledged this issue and has decided not to make a code change. Xena Finance states:

```
Contracts don't use this function. We keep it for compatibility with backend/frontend logic.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Wrong Accounting upon Margin Account Top up Specification Changed

Medium-Severity Findings	2

- Hardcoded Contract Addresses Code Corrected
- PriceReporter Will Not Execute Every Second Swap Order Code Corrected

Low-Severity Findings	5

- CEI Pattern Not Applied Code Corrected
- Incorrect Comments Specification Changed
- Interest Rate Is Not Constrained Code Corrected
- Missing Events Code Corrected
- IPool Is Missing Signature for liquidatePosition() Code Corrected

6.1 Wrong Accounting upon Margin Account Top up

Correctness High Version 1 Specification Changed

CS-XENA-001

When collateral is added to a long position without changing the position's size, the function `_reservePoolAsset` distributes the collateral in the tranches' `poolAmount` and `guaranteedValue`. The distribution is done according to the risk factor and current utilization of each tranche, calculated in `_calcReduceTranchesPoolAmount()`.

Note that the amount of collateral that can be added to a tranche is limited to the unreserved amount available in that tranche. In an extreme case where all tranches have a high utilization, it will be impossible to add a large amount of collateral while keeping position size equal, as `_calcReduceTranchesPoolAmount()` will revert if the amount of collateral to add is higher than the total unreserved amount in all tranches.

When releasing the asset, the distribution is done according to the ratio of reserved amounts across the tranches. In the case of a collateral top-up, collateral will be distributed among the tranches, but no additional amount is reserved. This means the distribution of the collateral and reserved amount may not match. This may lead to a wrong accounting, incorrect pricing of LP shares, and reverting transactions.

Consider the following example:

There are 2 tranches, each with the same risk factor for a given asset. When a long position is opened, tranche 0 is at full utilization, so the entire collateralAmount and reserveAmount will be given to tranche 1. Time passes and now the trader wants to increase their collateral, while keeping the size the same. At this point in time, tranche 1 has full utilization, so all the extra collateral will be given to tranche 0. No additional amount is reserved. Now, the trader closes their position. The full amount of collateral (that they deposited over 2 transactions) will be withdrawn from tranche 1's poolAmount, as only the ratio of reserveAmount is taken into account when closing a position. This is incorrect, as a part of the collateral was actually attributed to tranche 0. Tranche 1 will have fewer funds than it should, while tranche 0 will have more.

The `guaranteedValue` for the tranches will also be incorrect.

Specification changed:

Xena Finance acknowledged the issue and changed the spec to use only one tranche, which resolves the issue. Xena Finance stated:

```
We will use 1 tranche model for this version.
```

6.2 Hardcoded Contract Addresses

Correctness | Medium | Version 1 | Code Corrected

CS-XENA-003

In `LiquidityRouter`, the address of the wrapped ether contract is hardcoded.

```
IWETH public constant weth = IWETH(0x82aF49447D8a07e3bd95BD0d56f35241523fBab1);
```

The same is true in `OrderLens`:

```
address constant WETH = 0x82aF49447D8a07e3bd95BD0d56f35241523fBab1;
```

While this would be correct on Arbitrum One, this project is intended to deploy on Base Mainnet. On Base Mainnet, this address does not contain a deployment of WETH. As a result, the router will not work with WETH since the call to `deposit()` will fail and make the transaction revert.

Similarly, in `Oracle`, an address for a sequencer uptime feed is hardcoded.

```
/// @notice arbitrum sequence uptime feed
AggregatorV3Interface public constant sequencerUptimeFeed =
    AggregatorV3Interface(0xFdB631F5EE196F0ed6FAa767959853A9F217697D);
```

This feed is specific to Arbitrum One and does not function on Base Mainnet.

If the codebase is deployed with the current hardcoded addresses, no funds will be at risk since the system will not work at all.

Furthermore, in `OrderManager` an address for `EthUnwrapper` is hardcoded.

```
address public constant ETH_UNWRAPPER = 0x1730CdEe8f86272eBae2eFD83f94dd9D5D855EeD;
```

A contract exists at this address on Base Goerli. Since no contract has been deployed at this address on Base Mainnet, we cannot determine whether it is correct.

Code corrected:

The addresses are now given as an argument in the constructor, or in `initialize()` for contracts that will be used behind a proxy.

6.3 PriceReporter Will Not Execute Every Second Swap Order

Correctness Medium Version 1 Code Corrected

CS-XENA-006

In `postPriceAndExecuteOrders()`, when looping over swap orders, `i` is incremented twice. Once in the post-loop expression, and once in the loop body.

As a result, half the orders will be skipped.

```
for (uint256 i = 0; i < swapOrders.length; i++) {
    try orderManager.executeSwapOrder(swapOrders[i], payable(msg.sender)) {} catch {}
    unchecked {
        ++i;
    }
}
```

Code corrected:

`i` is now only incremented once per loop.

6.4 CEI Pattern Not Applied

Design Low Version 1 Code Corrected

CS-XENA-007

The checks-effects-interactions pattern that prevents reentrancy attacks is not followed in the function `executeLeverageOrder()`. The status of the order is set to `FILLED` only after the call to `_executeLeveragePositionRequest()`, which may send ETH to the owner with full `gas()`.

We do not see a direct attack vector through this reentrancy, but we recommend fixing this as a preventative measure.

Code corrected:

The code has been updated to first mark the order as `FILLED`, and then make the call to `_executeLeveragePositionRequest()`.

6.5 Incorrect Comments

Design Low Version 1 Specification Changed

CS-XENA-010

1. The comment above the function `OrderManager._createIncreasePositionOrder` specifies the construction of the `_data` field. It mentions a `uint256 collateral`, but no such field is decoded from the `_data`.

2. In the struct `DataTypes.Position`, the comment on the member `reserveAmount` says the amount is in `indexToken`, but the amount is denominated in `collateralToken`.
3. In `PoolV1.md`, the formula for long-side ManagedValue reads $(poolAmount - reserve) \times indexPrice - guaranteedValue$ it should read $(poolAmount - reserve) \times indexPrice + guaranteedValue$
4. Some of the comments describing the constants in `Oracle.sol` do not represent the correct units. For example:

```
MAX_CHAINLINK_TIMEOUT = 1 days; // 10%
```

Specification changed:

All mentioned comments have been corrected.

6.6 Interest Rate Is Not Constrained

Correctness Low Version 1 Code Corrected

CS-XENA-012

In `Constants`, an upper bound for the interest rate is provided:

```
uint256 public constant MAX_INTEREST_RATE = 1e7; // 0.1%
```

However, this value is not used anywhere. In particular, in `InterestRateModel`, there is no constraint on the interest rate parameter.

Code corrected:

The interest rate in `SimpleInterestRateModel` is now constrained to be strictly smaller than `MAX_INTEREST_RATE` in the constructor.

6.7 Missing Events

Design Low Version 1 Code Corrected

CS-XENA-013

1. When the `OrderManager` is initialized, the oracle is also set but no related event is emitted. This is not consistent with the `setOracle()` function, which emits an event.
2. When the `PriceReporter` adds and removes reporters, no event is emitted. This is not consistent with the similar functionality in `Oracle`, which emits events.

Code corrected:

1. The `OracleChanged` event is emitted at the end of `initialize`.
2. The events `ReporterAdded` and `ReporterRemoved` are emitted when a reporter is added/removed.

6.8 `IPool` Is Missing Signature for `liquidatePosition()`

Design | Low | Version 1 | Code Corrected

CS-XENA-016

The interface of the `Pool`, `IPool`, contains the events and errors relative to a liquidation, but is missing the signature of the function `liquidatePosition`.

Code corrected:

The missing function was added to the interface.

6.9 Duplicate Code

Informational | Version 1 | Code Corrected

CS-XENA-020

`LiquidityCalculator._calcDaoFee()` is never called and is a copy of `Pool._calcDaoFee()`.

Code corrected:

The function `LiquidityCalculator._calcDaoFee()` has been removed from the codebase.

6.10 Oracle Reporter Address Consistency

Informational | Version 1 | Code Corrected

CS-XENA-025

The function `Oracle.addReporter` allows the owner to add the `address(0)` as a reporter, but the function `Oracle.removeReporter` does not allow to remove the `address(0)`. This is not consistent with the behavior of the same functionality in `PriceReporter`, which does not allow adding `address(0)`.

Code corrected:

`Oracle.addReporter` no longer allows the `address(0)` to be added as a reporter.

6.11 Use of `assert()`

Informational | Version 1 | Code Corrected

CS-XENA-028

The function `Pool.setTargetWeight()` is using `assert(isAsset[item.token]);` to ensure that a token is in the mapping of assets.

The Solidity documentation states the following:

```
Assert should only be used to test for internal errors, and to check invariants.
Properly functioning code should never create a Panic, not even on invalid external input.
```

Moreover, failing assertions will consume all the remaining gas. This is why a `require()` statement should be used instead.

Code corrected:

The `assert` has been replaced by a `require` statement;

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 EVM Version

Informational **Version 1** Acknowledged

CS-XENA-021

When deploying on non-Ethereum chains, compatibility with the different EVM versions should be considered. It must be checked that the target chain supports the used Solidity version. For example, not every chain supports the `PUSH0` opcode, introduced in Solidity 0.8.20. If the Solidity version used is changed to something other than `0.8.18` in the future, these differences between chains should be considered.

7.2 Gas Optimizations

Informational **Version 1** Acknowledged

CS-XENA-022

The following is an incomplete list of possible gas optimizations:

1. Duplicated slippage protection: `OrderManager.executeSwapOrder()` implements slippage protection, but `Pool.swap()` already has the same functionality.
2. The field `SwapOrder.price` is assigned in `OrderManager.placeSwapOrder()` but never used.
3. The mappings `userLeverageOrderCount` and `userSwapOrderCount` are redundant, as a getter returning the length of the `userLeverageOrders` and `userSwapOrders` arrays would accomplish the same thing with higher gas efficiency.
4. The storage slot `SimpleInterestRateModel.interestRate` can be immutable.
5. The parameters `_minAmountOut` and `receiver` of the function `OrderManager._poolSwap` are always `0` and `address(this)`. They could be replaced by their fixed value to decrease the length of the calldata.
6. The check done in `OrderManager._requireControllerOrOwner` could be transformed following De Morgan's law to be more gas efficient and leverage the lazy evaluation of the parameters.
7. The function `OrderManager.cancelSwapOrder` could load only the specific fields of `order` that are needed into memory, instead of the whole struct, since not all the fields will be read. The same applies for `request` in `OrderManager._expiresOrder()`.
8. In `OrderLens.getOpenLeverageOrders()`, an array of constant size is first filled, then another array is created which contains the same elements, except that empty items are removed. Instead, only non-empty could be added to a dynamic size array using `array.push()`. This could avoid needing the second array. The same applies for `OrderLens.getOpenSwapOrders()`.
9. In most `for()` loops, incrementing the counter can be marked as `unchecked` to avoid an unnecessary overflow check. This is done in some places, but not systematically.
10. In certain callpaths, the oracle is queried multiple times for the same price. Caching certain prices could save gas. An example is the `addLiquidity` callpath, where the oracle is queried once for

each supported token from `calcAddLiquidity()`, then twice more from `refreshVirtualPoolValue()`.

11. Most `uint256` storage slots in `PoolStorage` have a known maximum size. For example, fee values or the accrual interval. These values could use smaller data types, which would allow packing them with other values to save gas.
12. The `SwapOrder` and `LeverageOrder` structs could be optimized by choosing a smaller data type for some of the fields, e.g., `submissionBlock`, `submissionTimestamp`, or `expiresAt`.
13. Upon interest accrual, it could save gas to check whether the interest has already been accrued for the current interval and return early if it has, avoiding the interest rate computation.

Acknowledged:

Xena Finance acknowledged this issue and has decided not to make a code change.

7.3 Hardcoded Contract Address

Informational Version 1

CS-XENA-023

In OrderManager, an address for EthUnwrapper is hardcoded.

```
address public constant ETH_UNWRAPPER = 0x1730CdEe8f86272eBae2eFD83f94dd9D5D855EeD;
```

A contract exists at this address on Base Goerli. Since no contract has been deployed at this address on Base Mainnet, we cannot determine whether it is correct.

7.4 Misleading `onlyController` Name

Informational Version 1 Acknowledged

CS-XENA-024

The `onlyController/_onlyController` modifier/function's name is misleading, as they will accept the `owner` address as well, not only the `controller`.

Acknowledged:

Xena Finance understands and acknowledges the issue.

7.5 Tokens With Low Decimals

Informational Version 1 Acknowledged

CS-XENA-026

All computations that split token amounts into different ratios are rounded down, which is safer for the system. While it is not a problem for tokens with enough decimals, it could impact tokens with low decimals (e.g., GUSD has 2 decimals) or relatively low decimals compared to its value (e.g., WBTC has 8 decimals). Such rounding could result in value being locked in the contract.

Tokens used must be carefully chosen so the value lost in precision errors is not too high.

Acknowledged:

Xena Finance understands and acknowledges the issue.

7.6 Tokens With Many Decimals

Informational Version 1 Acknowledged

CS-XENA-027

As the normalized price in `Oracle` is stored as a 30 decimal USD value, tokens used must be carefully chosen so their price will have enough precision.

For example, a token with a number of decimals close to 30 will have a low price precision.

Acknowledged:

Xena Finance understands and acknowledges the issue.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Add Then Remove Liquidity May Be Cheaper Than Swap

Note Version 1

Adding or removing liquidity, as well as swapping, has a fee. These fees are separately configured.

For a swap, the fees for in and out token are calculated and the higher fee is used. For liquidity, there is a fee on add as well as on remove.

If the fees for liquidity adding/removing are sufficiently low, it may be cheaper to emulate a swap by first adding and then removing liquidity in another token, than it is to do a direct swap. In particular, there is a minimum fee that is enforced on swaps, but this minimum is not enforced on add/remove liquidity.

8.2 LPs Are Not Always Able to Remove Liquidity

Note Version 1

Liquidity removal from a tranche is only possible within the unreserved amount, i.e. `poolAmount - reservedAmount`, so if the tranche is fully reserved or the LP has a position in a tranche greater than the unreserved amount, removal of liquidity will be limited to that amount.

LPs do not have a way to force the closure of open positions, so they may be unable to withdraw for a long time in this situation. However, they will also be earning a high interest rate.

8.3 Market Manipulation of Listed Assets Must Not Be Profitable

Note Version 1

Xena relies fully on an external oracle to determine the prices it offers to traders. This is what enables the zero-price-impact trading feature.

However, it also comes with significant risks. Unlike spot markets, large positions can be opened and closed without affecting the market price. This means that it is important that the price on the market from which the external oracle gets its prices must not be manipulatable by an attacker.

If an attacker is able to move the external market price by an amount larger than the fee paid to swap or open and then close a position in Xena combined with the cost of the manipulation, this could be a profitable attack. The loss from such an attack would be taken by LPs. The profit for the attacker is limited by the percentage they manipulate the market and the maximum size of a position (or swap) the attacker can create.

If the attacker keeps their position open for less than `accrualInterval`, they may not need to pay any interest. See Free Leverage Within Accrual Interval.

To mitigate this attack vector, Xena has 2 features: A `maxLiquidity`, which limits the size of longs and swaps, and a `maxGlobalShortSize`, which limits the size of shorts. Each of these can be configured per token.

These values must be configured carefully, in such a way that the cost of manipulating the external market is always larger than the profit that can be made from exploiting projects relying on that market's price. Only assets with highly liquid markets should be listed on Xena. The less liquid the external market is, the lower the `maxLiquidity` and `maxGlobalShortSize` must be. If an asset becomes less liquid over time, it should be delisted, or the limits should be lowered. Additionally, the limits should be lowered if there is another project (for example another deployment of Xena) that also relies on the same asset's price. In this case, the limits must be coordinated such that the total profit among all projects from manipulating the price is still smaller than the cost of manipulation.

A historical example of such an attack on another zero-price-impact DEX (GMX) can be found here: https://web.archive.org/web/20221015123657/https://twitter.com/joshua_j_lim/status/1571554171395923968

8.4 Outdated Virtual Pool Value

Note Version 1

If liquidity is not added or removed frequently, the virtual pool value may be outdated. Integrators relying on the virtual pool value should call `refreshVirtualPoolValue` before using the value.

8.5 Pay Token and Returned Token May Differ

Note Version 1

When using `ETH` or `WETH`, users must be aware of the following behaviors:

- If the `tokenIn` of a swap order was `ETH`, then `WETH` will be transferred back to the owner if the order is cancelled.
- If the `payToken` of a leveraged order was `WETH`, then `ETH` will be transferred back to the owner if the order is cancelled.
- If the `payToken` of a leveraged order was `WETH`, then `ETH` will be transferred back to the owner if the order expires.

8.6 Public Execution of Leverage Orders Does Not Work if Keeper Is Down

Note Version 1

The `publicExecution` flag should allow order owners to execute their own orders, even when the executor is inactive.

However, public execution of leverage orders will not work if the Keeper does not post prices anymore. Leverage orders can only be executed if a price update happened after it was placed.

8.7 Senior Tranche Assumes Full Risk in Extreme Situations

Note Version 1

Different tranches have different risk exposures, which are dependent on the `riskFactor` of the tranche.

In normal circumstances, a tranche will only assume a percentage of risk of each trade according to their `riskFactor`. However, in extreme scenarios where the other tranches are already fully utilized, the senior tranche can be exposed to `100%` of the risk of a trade. These extreme situations likely have a higher risk to the LP compared to "normal conditions". This leads to senior tranches having lower risk exposure (and lower fee revenue) during "normal conditions" and full risk exposure during "extreme conditions". This may lead to the overall `risk/reward` ratio for senior tranches to be worse than for junior tranches.

Users should take this into account when deciding which tranche they want to participate in.

Xena Finance affirmed that they are aware of this and that it is the intended behavior.

8.8 Swaps on `Pool` Should Be Done Atomically

Note Version 1

When users are directly using the `swap` function of the `Pool`, it must be done within one transaction. The `swap` function expects the user's funds to already have been transferred to the contract before the call. If users were to first send the funds and then call `swap()` in two separate transactions, they will likely be front-run and lose their funds.

The `OrderManager` and `LiquidityRouter` provide helper functions to transfer and swap in the same transaction.

8.9 System Is Paused if Chainlink Is Down

Note Version 1

The system relies on Chainlink prices for every user action. Users must be aware that the system will not work if one of the following conditions is met:

- the Chainlink price has not been updated for some time and is stale (every token has a configurable timeout)
- The chain's (initially Base) sequencer is down according to the Chainlink sequencer uptime feed
- The chain's sequencer restarted less than 1 hour ago according to the Chainlink sequencer uptime feed

8.10 Transaction Ordering MEV

Note Version 1

Transaction ordering in Xena is very important. Similarly to other markets, the execution of a transaction depends on the transactions before it. For example, the fees charged for a swap are dependent on the current token ratios in the pool. This is also known in the context of Ethereum as MEV.

In Xena, any user can make a `swap` at any time. However, other order types can only be executed by the `executor` role while `publicExecution` is disabled. The smart contracts do not enforce any transaction order, so the `executor` is free to decide in which order they execute orders. It can also decide to censor some orders, never executing them. This is comparable to the role of block builders in Ethereum.

The `executor` has the potential to use its privileged position to extract some of this MEV-comparable value that is present for itself. Additionally, it seems to be intended that the `executor` role is held by the `PriceReporter`, which also has the powerful role of providing asset prices to the `Oracle`. Using `postPriceAndExecuteOrders`, the `PriceReporter` can update the oracle price and then immediately execute orders. In particular, it can execute `swap` orders immediately, before other addresses have a chance to `swap` using the updated prices.

Users must ensure that the `executor` and `PriceReporter` are behaving as expected and are not using their privileged position to extract value for themselves, for example by taking payments for quicker execution, censorship, or by executing their own transactions first.

The `executor` should publish its transaction ordering methodology, so that users can hold it accountable if it does not behave accordingly. It would also be possible to enforce some ordering rules on the smart contract level.