PUBLIC

Code Assessment

of the Kasu Core Smart Contracts

October 01, 2024

Produced for

KASU

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Kasu team,

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

Kasu implements a lending protocol that allows users to deposit USDC into pools and earn interest. Pool fund managers can use the funds to invest off-chain and pay back the loans on-chain to the protocol users. The code in this review added the functionality for fix term deposits.

As before, the communication with the developer team was always professional and questions were answered quickly. The documentation is good and comprehensive. The code is written very straightforwardly and well-structured. In some cases, gas efficiency seems to be sacrificed for readability and comprehensiveness. Especially, the newly added fix term deposit feature could be improved in terms of gas efficiency.

We did not uncover any severe issues in this iteration. All previously issues raised, are resolved or acknowledged.

In summary, we find that the codebase provides a high level of security. Yet, 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	0
Medium-Severity Findings	1
• Code Corrected	1
Low-Severity Findings	8
• Code Corrected	7
• Acknowledged	1

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 smart contract source code files inside the Kasu Core 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	17 April 2024	b1b989776733c067a4d6b8983d52778cb8fa809d	Initial Version
2	12 May 2024	5e2b70d3b90172a1422192b1aa95522a01bf1e2b	Version 2
3	24 May 2024	0927bf63f68a0cd0dda9da6bc1d123371f5c3aec	Version 3
4	02 July 2024	7129503b73dbefb4d3ac169991cec6611ab45a20	Version 4
5	03 July 2024	43472ab937183406ed9aec6533b3f08e8fcd5de8	First public report
6	19 August 2024	c4ffc80e81b3426ae424d827a120c1fc826dfcc1	Fixed-term deposit
7	13 September 2024	3cc8b0ce18837b2cc2e975411dc3eeb1b3442ed3	Version 5

For the solidity smart contracts, the compiler version `0.8.23` was chosen.

2.1.1 Excluded from scope

- Third-party libraries and integrations
- Mock contracts and test-related code

The report was extended with additions in Version 4. The following functionality was mainly added:

- additional tracking of the number of tranches a user is invested in
- an admin function to set the number of tranches manually that also enables/disables user reward eligibility
- only users who deposited are eligible for rewards

The affected contracts are: `LendingPoolTranche`, `UserManager` and `KSULocking`.

In Version 6. The fixed term deposit functionality was implemented in `FixedTermDeposit` and additionally affected mainly the following contracts: `LendingPool`, `LendingPoolManager`, `PendingPool`, `LendingPoolTranche` and `ClearingCoordinator`.

2.2 System Overview

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

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

Kasu offers a lending protocol allowing a pool manager to use the users' funds for off-chain (so-called Real World Assets or short RWA) investments. Users wishing to provide liquidity (USDC) to the system have the option to choose a pool and a risk profile (known as tranches). The pool manager can withdraw the funds and use them off-chain. After successful or unsuccessful investments, the pool manager pays the funds back into the pool or reports a loss. Users can at any time withdraw funds if sufficient funds are held as a buffer on-chain. Otherwise, they need to wait for funds to be repaid by the pool manager. Users need to be whitelisted and not blacklisted to invest. Investments cannot be done directly by the users. They first need to request a deposit that will be processed once in an epoch. If a user investment is accepted depends on the available amount in the tranches and the priority level of a user. The priority level is calculated as the ratio between the funds a user wants to invest and staked KSU token. Withdrawals need to be requested too. The priority of a withdrawal will increase to the highest possible (after admin-initiated withdrawals) after five attempts. In case of emergency, admins can set the highest possible priority for certain requests (called system requests) or even directly and immediately draw the funds out of the pool back to the user. Kasu charges two types of fees to liquidity providers. Ecosystem fees are sent to the KSU Locking contract and distributed to rKSU holders, while protocol fees are stored in the contract until claimed.

2.2.1 KSU Tokens

KSU token is the native token of Kasu. Users holding KSU tokens can lock them for a period of time and get rKSU. The amount of received rKSU depends on the locked KSU and the duration of locking. Locking KSU has some benefits:

1. Users holding rKSU participate in the protocol fees.
2. The ratio between rKSU balance to provided liquidity of a user determines his loyalty level, hence the priority of his request
3. Bonus locking program rewards KSU lockers with additional tokens
4. Additional lending APY for the liquidity providers

At the beginning, `1_000_000_000 * 10**18` KSU tokens are minted. Thereafter, no more KSU tokens can be minted.

2.2.2 Epochs

Kasu's progress is divided into a series of epochs. Each epoch can be divided into the following stages (phases):

- Start of an epoch
- KSU token pricing
- Receiving deposit/withdrawal requests
- Clearing phase
- End of the epoch

Users can submit their requests at any time. However, if the request will be batched at executed at the end of an epoch and after the finalization of the clearing process. If a request is submitted during the clearing phase of an epoch, its effective epoch would be the subsequent epoch.

2.2.3 LendingPoolManager

The main entry point to almost all functionality is the `LendingPoolManager` contract. It provides the functionality to:

- create a pool to collect funds from users and use these funds for real-world investments (see Investment pools). (`createPool()`).
- request deposits and withdraws or cancel them (when still possible) for users. (`requestDeposit()`, `requestDepositWithKyc()`, `cancelDepositRequest()`, `requestWithdrawal()`, `cancelWithdrawalRequest()`).
- repay the owed funds for pool fund managers. (`repayOwedFunds()`).
- report a loss or repay a loss for pool fund managers (`reportLoss()`, `repayLoss()`).
- to claim users eventually repaid funds from a loss that was reported (`claimRepaidLoss`).
- change the address to which funds can be withdrawn for the pool admin (`updateDrawRecipient`).
- deposit and withdraw unused first loss capital for the pool fund manager (`depositFirstLossCapital`, `withdrawFirstLossCapital`).
- do the periodical clearing by the pool clearing manager (`doClearing`).
- immediately force a user(s) withdrawal for the pool manager (`forceImmediateWithdrawal`, `batchForceWithdrawals`).
- force cancel a user request for the pool manager (`forceCancelDepositRequest`, `forceCancelWithdrawalRequest`)
- to stop a lending pool at the end of its lifecycle for the pool manager (`stopLendingPool`)
- to manage pool parameters for the pool manager (`updateTargetExcessLiquidityPercentage`, `updateMinimumExcessLiquidityPercentage`, `updateMinimumDepositAmount`, `updateMaximumDepositAmount`, `updateTrancheInterestRate`, `updateTrancheDesiredRatios`, `updateDesiredDrawAmount`, `updateTrancheInterestRateChangeEpochDelay`)

All the above-mentioned functions usually have very limited implementation in the `LendingPoolManager`. They act as access control and forward the call to the respective contract that contains most of the relevant logic and state.

2.2.4 Investment pools

The core of the system is investment pools for users to invest in. A pool consists of multiple contracts:

- a pending pool that manages the pending request to deposit or withdraw funds from the pool. It issues an NFT to a user with the details of their request.
- a lending pool that manages the funds that were approved to be invested. The lending pool issues pool shares for a deposit and burns them in case of a withdrawal. The pool shares are minted and deposited into a tranche. A tranche is an ERC4626 vault representing different risk levels for investors. It also contains the logic for repayment and to manage losses (reporting a loss and re-payment of losses).
- one or max three tranche contracts representing the different risk levels possible (junior, mezzanine, and senior, in the decreasing order of risks and yields).

All contracts form an investment pool and are deployed together via a factory contract by a special `ROLE_LENDING_POOL_CREATOR` role. Hence, Kasu can have multiple lending pools, pending pools and tranche contracts but each in a set forming one investment pool.

2.2.5 Pending Pool

Users willing to participate in the Kasu ecosystem submit their requests to the pending pool. A pending pool temporarily takes custody of users' deposits and mints ERC721 tokens as a receipt for them. Users can also withdraw their capital from the Kasu ecosystem by submitting their withdrawal requests. The details of a user's deposit or withdrawal request will be saved together with a non-transferrable NFT position representing the request. While the system is in the clearing phase of an epoch, the clearing process can be initiated by the lending pool manager to determine which funds should be accepted or forwarded to another tranche. Pending pools have the functionality to manage the deposit and withdrawal requests like:

- request a deposit or withdrawal (`requestDeposit`, `requestWithdrawal`)
- cancel a deposit or withdrawal request (`cancelWithdrawalRequest`, `cancelDepositRequest`)
- force cancel a deposit or withdrawal request (`forceCancelDepositRequest`, `forceCancelWithdrawalRequest`)
- initiating a batch of forced withdrawals for users (`batchForceWithdrawals`)

All these functions are restricted to be called from the `LendingPoolManager` contract. Additionally, a `stop` function can be called by the lending pool manager to stop a whole pool (lending, pending and tranche contracts). When funds are accepted in the clearing phase, they will be forwarded to the corresponding lending pool.

Liquidity providers should first **request** a deposit into their system. Users can send their ETH or tokens of interest to the Lending Pool Manager. As Kasu Core' supported currency in USDC, tokens other than USDC as well as ETH will be routed to a swapper, which swaps them to get USDC. Afterward, these deposited USDC will be transferred to the Pending Pool. Upon receiving the pending deposit, the Pending Pool mints a Deposit NFT (DNFT) encoding the following information for the liquidity provider:

1. Amount
2. Tranche of interest
3. The request epoch

Then, the user should wait until clearing is finalized and the system decides whether to accept this request (fully or partially) or reject it. Until clearing, a user can submit further deposit requests and get the respective DNFT. But, if they request to deposit into a tranche which they have already deposited, he does not receive a new DNFT.

Before clearing starts, the users have the opportunity to cancel their deposit request. This procedure decreases the requested amount represented by the DNFT (and burns it if the requested amount reaches 0). The pool manager can also request to cancel withdrawals on behalf of users.

The clearing process tries to forward the deposited amount to the tranche of interest. However, if this tranche cannot accept the total deposit of the user, a part of it, gets accepted, while the rest would get forwarded to the lower tranches. Finally, if there are still some deposited funds that the system fails to deposit to any tranche, they will get rejected and sent back to the liquidity provider. When a tranche accepts a deposit from a user, ERC20 tokens get minted to represent the user's share in the tranche for the liquidity provider.

Whether a deposit gets accepted or rejected, its corresponding DNFT gets burned.

After successfully depositing their liquidity into the system, liquidity providers can also request to withdraw their funds. For this purpose, firstly the tranche share tokens are sent to the pending pool. Then, similar to depositing, an NFT (WNFT) gets minted. This request stays pending until clearing and in between, users can

- increase the desired amount of withdrawal
- cancel their request

Pool managers can submit forced withdrawals with the highest priority in the system, namely `SYSTEM`, and the withdrawal request submitted by the users gets `USER` priority. Later, during clearing, `USER` withdrawal levels get priority: 1. If a request is pending for longer than 5 epochs, it gets the highest `USER` priority or 2. if not, the priority is the same as the loyalty level of the user.

After the clearing, the accepted withdrawal requests get executed, funds are transferred to the user and the tranche token gets burned.

2.2.6 Lending Pool

Lending pools are the core contract that:

- handle the deposit and withdrawal of funds and the corresponding pool tokens (`acceptDeposit`, `acceptWithdrawal` and `forceImmediateWithdrawal`)
- hold the active funds until they are drawn and invested off-chain by the fund manager (via `drawFunds`)
- mint new IOUs representing the interest to the corresponding tranches (`applyInterests`)
- manage parts of the fee payments (`payOwedFees`)
- manage the repayment of the funds by the fund manager (`repayOwedFunds`)
- allow the deposit and withdrawal of first loss capital (`depositFirstLossCapital`, `withdrawFirstLossCapital`)
- manage potential losses, and their repayment and issue NFTs to affected users that represent a claim on future repaid losses(`reportLoss`, `repayOwedFunds` and `claimRepaidLoss`)

When funds from a user deposit request are accepted, they are sent from the pending pool to the lending pool. The lending pool will receive the USDC and mint a 1:1 IOU represented as a pool token. The pool tokens are then forwarded to the corresponding tranches. Each tranche is an ERC4626 vault that issues non-transferrable vault shares to the users.

The pool funds manager determines the recipient of the drawing during the clearing. They can later repay the owed amount from a repayment address. It first transfers the funds to the pending pool and then gets forwarded to the lending pool. After receiving the funds, a part of them gets deducted as fees. Fees consist of two parts: 1. ecosystem fee which gets locked in the KSU locking contract and increases the rewards per share. The rest gets accumulated to the protocol fee.

If after drawing the funds, the draw recipient encounters a loss, the Pool Funds Manager can report the loss to the lending pool. The lending pool tries to cover this loss in the following order:

1. First loss capital
2. Junior tranche
3. Mezzanine tranche
4. Senior tranche

And burns the corresponding lending pool tokens. If the first loss capital is not enough to cover the losses, the affected tranches mint their own ERC1155 loss tokens. When a loss is realized, the user owed amount gets decreased as well.

After getting reported, each loss gets its own loss ID. This loss ID can later be used to repay the loss.

If the Pool Funds Manager manages to receive funds (e.g. from a liquidation), they can pay back the amount to the users. The funds will be forwarded to the affected tranche contracts. Users who have deposited in this affected tranche and received a loss NFT, can claim their loss depending on (1) how much the Pool Funds Manager has repaid (2) what was their initial share of tranche tokens.

The lending pool is stopped together with the associated pending pool over the `LendingPoolManager` contract. In order to be stopped, the user and fee owed amount should be `0`. When stopped, the lending

pool cannot accept any further deposit or withdrawal requests. Once stopped, it can never resume working again.

2.2.7 Tranches

Tranches represent different risk profiles associated with a repayment priority in the event of an unrealized loss. Each tranche is represented by its own tranche contract that is deployed when a new investment pool is created. The tranches are dedicated to one lending pool.

Lending pools in Kasu support up to 3 tranches:

1. Junior: offers the highest interest. However, in case of any loss, the investors in the junior tranches are on the frontline of absorbing the loss.
2. Senior: takes a conservative approach by offering the lowest interest but has the highest claim to being repaid, prioritizing security over return.
3. Mezzanine: offers a middle ground, providing a moderate interest and risk level.

2.2.8 Clearing Process

Clearing will mainly process the deposit and withdrawal requests. To do so, it will calculate the pool interest, the deposit and withdrawals that will be accepted, provide the desired funds to the fund manager and pay outstanding fees, if possible.

Before clearing starts, the loyalty level of users should be calculated (to assign them their corresponding priorities). The 5 main stages of the clearing process are:

1. Applying pool interest to the tranches: Lending pool tokens of USDC IOU are minted and transferred to the lending pool's tranches respective to each tranche's interest rate.
2. Calculating request priorities: users with higher loyalty levels get a higher priority for their deposit request while for withdrawals, apart from the loyalty level, two other factors play a role:
    1. Withdrawals forced by the system get the highest priority.
    2. Requests pending for longer than 5 epochs get the second highest priority.
3. Accepting/Rejecting requests: By considering the desired ratio for distribution of funds among tranches and the desired draw amount of the fund's manager as well as priorities of the requests, the system finds out which request to accept and which to forward to a lower-level tranche. For example, if the requested draw amount exceeds the total available excess funds and total deposit, the clearing fails.
4. Finalize the accepted requests: once the system decides which requests to accept and which to reject, it loops over the accepted deposit/withdrawal requests. When a user's request is finalized, his NFT minted by the pending pool upon initiating the request gets burned.
5. Draw funds: A predefined desired amount of funds goes to a defined recipient.

If a Liquidity Provider demands a specific tranche that is oversubscribed, additional orders for that tranche are automatically redirected to the next available tranche with available capacity. This mechanism ensures that capital is kept active, and that Liquidity Providers still have the opportunity to participate.

2.2.9 UserLoyalityRewards

This contract calculates the KSU rewards for the users based on their loyalty levels and allows the users to claim their rewards via `claimReward`. It also has a sweep function to recover eventually stuck tokens that can be called by the admin (`recoverERC20`).

2.2.10 KSU Token And Locking

KSU holders can lock their KSU token via `lock()` or `lockWithPermit()` for a certain time. Depending on the time the user decides to lock their tokens, they will receive a reward and (until depleted) bonus tokens from the `KSULockBonus` contract. The locked USD token value of a user is crucial for the loyalty level calculation. The loyalty level is the ratio between the value of the locked KSU token and the value of the USDC the user has deposited and is requesting to deposit into a Kasu lending pool. Locked KSU tokens are represented by non-transferrable rKSU tokens.

Users can unlock their tokens when the lock time has passed. The admin has additional functionality like adding locking periods or the option to immediately withdraw on behalf of a user by calling `emergencyWithdraw`.

2.2.11 UserManager

This contract keeps track if users are registered and what lending pool they interact with. It also keeps track of the user's loyalty level (and associated details) per epoch. The loyalty level is calculated based on the amount of KSU tokens a user holds and the amount of USDC a user has deposited into the pool. The loyalty level is used to determine the priority of a user's deposit or withdrawal request.

2.2.12 Roles & Trust Assumptions

The following roles are fully trusted due to their power in the system:

- Kasu contract upgrader (proxy admin)
- ROLE_LENDING_POOL_CREATOR
- ROLE_POOL_FUNDS_MANAGER
- ROLE_POOL_MANAGER
- ROLE_POOL_CLEARING_MANAGER
- ROLE_POOL_ADMIN
- ROLE_KASU_ADMIN
- ROLE_SWAPPER (should be the `DepositSwap` contract)
- ROLE_PROTOCOL_FEE_CLAIMER (limited impact if not claimed)

We further assume that:

- Oracle contract always provides the correct price.
- Third-party code used in the code base is safe (e.g., OpenZeppelin contracts).
- Exchanges are tested and carefully added to be non-malicious.
- Only USDC is used as the underlying asset.
- The Know Your Customer (KYC) signature provider is trusted.

2.2.13 Changes in Versions

In **Version 6**, the option for lender to lock their assets for a fixed period for a fixed yield was added to the protocol. This feature is mainly implemented in `FixedTermDeposit` contract.

The Pool Manager can add fixed-term deposits configuration for a specific lending pool and tranche. The configuration specifies the epoch interest rate, the lock duration in epoch as well as wether the configuration is based on a whitelist or not.

Given a fixed-term configuration, users have two ways of locking their assets:

1. When requesting a deposit, the user can specify a configuration ID, upon acceptance of the deposit, the user's assets are locked for the specified duration. It should be noted that only the assets going into the requested tranche are locked, the assets allocated in tranche with lower risk profile are not locked (spillover funds will not be locked).
2. Given some existing deposit, the user can lock their tranche's shares by calling `lockDepositForFixedTerm`.

In practice, locking the assets is done by transferring the user's tranche shares to the `FixedTermDeposit` contract. At the end of the lock duration, the shares are transferred back to the user automatically (when clearing is done). When users flag they want to withdraw their tranche shares after the fix term lock ended, the underlying is transferred to the users. The default case if a user does not signal this, locked shares will be send back to the tranche as usual. Users need to signal this full withdrawal before a specified deadline is passed (canceling this request can also only be done until a certain deadline). When the lock period is over and a user signaled the full withdraw including redeeming from the tranche, the user's request is processed the highest priority.

During clearing, the interest are applied as follow:

1. Apply interest as usual based on the base APY, mint and transfer IOU tokens to the tranches accordingly.
2. Apply the fixed term interests as follow:
    - If the fixed term deposit APY is higher than the base APY, additional IOU tokens are deposited to the respective tranche for the delta between the two APYs. The obtained tranche shares are transferred to the `FixedTermDeposit` contract and accounted to the specific user.
    - If the fixed term deposit APY is lower than the base APY, some of the user's locked shares are redeemed (from the `FixedTermDeposit`) and the obtained IOU are burned for the delta between the two APYs.

The Pool Manager can also cancel a fixed-term deposit at any time, in which case the user's shares are transferred back to the user.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	0
Low-Severity Findings	1

- Inconsistent Update of Withdrawal Configuration Possible Acknowledged

5.1 Inconsistent Update of Withdrawal Configuration Possible

Correctness Low Version 6 Acknowledged

CS-KASU-Core-017

With `FixedTermDeposit.updateLendingPoolWithdrawalConfiguration` the time until a request must be submitted or canceled to be immediately valid can be set. This can even be done when the cancel request and request epoch started and some users have already requested or canceled a their withdrawal request.

Acknowledged:

The issue is acknowledged and Kasu further states that it is *up to the PD to set this configuration in time, and updating the variables only affects the users who request actions afterward*.

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	0
Medium-Severity Findings	1

- Potentially Missing Slippage Protection Code Corrected

Low-Severity Findings	7

- Incorrect Check When Canceling a Fixed Term Deposit Withdrawal Request Code Corrected
- The Initializer Is Not Disabled for FixedTermDeposit Code Corrected
- Event Defintions Code Corrected
- Infinite Exchange Approvals Code Corrected
- Non-disabled ERC1155 Function Code Corrected
- Spurious Stop Event Code Corrected
- Unclear Access Control Code Corrected

Informational Findings	4

- Inconsistent or Redundant Role Actions Code Corrected
- Incomplete NatSpec Code Corrected
- Cancel Incompletely Processed Withdrawal Request Specification Changed
- Typos Code Corrected

6.1 Potentially Missing Slippage Protection

Security Medium Version 1 Code Corrected

CS-KASU-Core-012

If a user deposits ERC20 tokens other than USDC or ETH, they can swap their token to USDC using the in-build swapper. Currently, the swapper has no slippage protection. Some exchanges might offer slippage protection by including it in the calldata object. But as the swapper is generic, Kasu might consider slippage protection on protocol level.

Code corrected:

Kasu has introduced `minAmountOut` to the `SwapDepositBag`. After swapping, it checks that the output amount is not less than `minAmountOut`:

```
outTokenAmount = IERC20(outToken).balanceOf(address(this));
```

```
if (outTokenAmount < swapDepositBag.minAmountOut) {
    revert InsufficientOutputAmount(outTokenAmount, swapDepositBag.minAmountOut);
}
```

6.2 Incorrect Check When Canceling a Fixed Term Deposit Withdrawal Request

Correctness | Low | Version 6 | Code Corrected

CS-KASU-Core-018

In `FixedTermDeposit`, the function `cancelFixedTermDepositWithdrawalRequest` checks if the user can still cancel the request with the following code:

```
_verifyWithdrawalActionTime(
    currentEpoch,
    deposit.epochUnlockNumber,
    _lendingPoolWithdrawalConfiguration[lendingPool].requestEpochsInAdvance
)
```

However, the duration provided to perform the check is `requestEpochsInAdvance` and not `cancelRequestEpochsInAdvance` as one would expect.

The issue appears as well when reverting with `FixedTermDepositWithdrawalRequestCancelTooLate` event:

```
revert FixedTermDepositWithdrawalRequestCancelTooLate(
    lendingPool,
    fixedTermDepositId,
    _lendingPoolWithdrawalConfiguration[lendingPool].requestEpochsInAdvance,
    deposit.epochUnlockNumber,
    currentEpoch
);
```

Code corrected:

Instead of `requestEpochsInAdvance`, `cancelRequestEpochsInAdvance` is now used.

6.3 The Initializer Is Not Disabled for FixedTermDeposit

Security | Low | Version 6 | Code Corrected

CS-KASU-Core-019

In `FixedTermDeposit`, OpenZeppelin's `Initializable` contract is used without calling `_disableInitializers()` in the constructor.

Code corrected:

The constructor was updated to call `_disableInitializers()`.

6.4 Event Defintions

Design Low Version 1 Code Corrected

CS-KASU-Core-007

Some event fields can be indexed to ease looking them up:

- UserAddedInAllowList(address user)
- UserRemovedFromAllowList(address user)
- UserBlockedFromAllowList(address user)
- UserUnblockedFromAllowList(address user)
- ProtocolFeeReceiverUpdated(address receiver)

Code corrected:

Kasu has correctly defined the aforementioned event parameters as indexed.

6.5 Infinite Exchange Approvals

Security Low Version 1 Code Corrected

CS-KASU-Core-013

The `Swapper` contract grants an infinite approval to an arbitrary `swapTarget` address set by the user requesting to deposit. There is no way to remove the approval nor is it removed immediately. The allow list only checks the `swapTarget` but cannot control the exact input and output tokens. Third party contracts might change over time and could even become malicious. Although it is assumed that the swapper should not have any tokens neither ETH, Kasu might reconsider what benefits and risks an infinite non-revokable (without upgrading the contracts) approval has.

If gas costs are a concern, maximum approvals should only be given when trusted contracts are called. E.g., from the `LendingPoolManager` to the pending pool to deposit USDC.

As Kasu can update almost all contracts, it might be dangerous if new functionality is added but not considered, that there are many infinite approvals pending.

Code corrected:

Kasu has added the following

```
_resetApproval(IERC20(swapInfo[i].token), swapInfo[i].swapTarget);
```

after calling into the swapper.

6.6 Non-disabled ERC1155 Function

Correctness Low Version 1 Code Corrected

CS-KASU-Core-016

The `LendingPoolTrancheLoss` contract disables the functions `safeTransferFrom` and `setApprovalForAll` but not the `safeBatchTransferFrom` function.

Code corrected:

This issued has been addressed by overriding

```
function safeBatchTransferFrom(address, address, uint256[] memory, uint256[] memory, bytes memory)
public
pure
override(ERC1155Upgradeable, IERC1155)
{
    revert NonTransferable();
}
```

6.7 Spurious Stop Event

Correctness Low Version 1 Code Corrected

CS-KASU-Core-010

The function `stop` could be called multiple times and emit a `LendingPoolStopped` event as there is no check if the lending pool is stopped already.

Code corrected:

`lendingPoolShouldNotBeStopped` modifier has been added to `stop()`, which avoids calling `stop()` multiple times.

6.8 Unclear Access Control

Design Low Version 1 Code Corrected

CS-KASU-Core-011

The function `FeeManager.emitFees` is only called by the `LendingPool` but has no access control. A caller would transfer assets into the contract. There seems no reason for anyone to call this function except in case of a malicious action. We cannot find that there is an attack vector but as there should be no incentive to call the function anyway, access control might be more safe.

Code corrected:

`FeeManager.emitFees()` assures that only the lending pool is calling it:

```
if (!_lendingPoolManager.isLendingPool(msg.sender)) {
    revert ILendingPoolErrors.InvalidLendingPool(msg.sender);
}
```

6.9 Incomplete NatSpec

Informational Version 6 Code Corrected

CS-KASU-Core-020

In `LendingPoolManager`, the following functions are missing NatSpec comments:

- `endFixedTermDeposit` is missing the `arrayIndex` parameter description.

- `addLendingPoolTrancheFixedTermDeposit` is missing the return value description.

Code corrected

The nat-spec comments were added to the functions in the code.

6.10 Cancel Incompletely Processed Withdrawal Request

Informational | Version 1 | Specification Changed

CS-KASU-Core-014

A not completely filled withdrawal request can be canceled after clearing happens as long as the new period has not started yet. This seems inconsistent with the specification.

Specification changed:

Kasu has explicitly mentioned that it is an expected behavior, replying:

> "This is expected. After the clearing is performed on the lending pool, so the clearing for the lending pool is not pending, it's possible to cancel the withdrawal request even if the clearing period is active."

6.11 Inconsistent or Redundant Role Actions

Informational | Version 1 | Code Corrected

CS-KASU-Core-009

The `KasuController` has the `ROLE_KASU_ADMIN` role which can set the boolean flag `$._paused` to pause certain functions. These functions have the `whenNotPaused` modifier that calls the controller to check the flag. Some of the functions with the `whenNotPaused` modifier also have the `onlyAdmin` modifier:

KasuAllowList

- setNexeraIDSigner() external whenNotPaused onlyAdmin
- allowUser() external whenNotPaused onlyAdmin
- disallowUser() external whenNotPaused onlyAdmin
- blockUser() external whenNotPaused onlyAdmin
- unblockUser() external whenNotPaused onlyAdmin

SystemVariables

- setPerformanceFee() external whenNotPaused onlyAdmin
- setLoyaltyThresholds() external whenNotPaused onlyAdmin
- setUserCanOnlyDepositToJuniorTrancheWhenHeHasRKSU() external whenNotPaused onlyAdmin
- setDefaultTrancheInterestChangeEpochDelay() public whenNotPaused onlyAdmin
- setMaxTrancheInterestRate() public whenNotPaused onlyAdmin
- setFeeRates() external whenNotPaused onlyAdmin

- setProtocolFeeReceiver() public whenNotPaused onlyAdmin

KsuLocking

- setKSULockBonus() external whenNotPaused onlyAdmin
- addLockPeriod() external whenNotPaused onlyAdmin

The same admin account manages all the functions including pausing. Consequently, limiting the above admin functionality with a flag that the admin can set is ineffective. It would only make sense to avoid mistakes. But it even seems reasonable to e.g., maintain the blocklist in a paused system. We encourage Kasu to check if the behavior is intended and give us feedback on the reasoning behind this behavior.

Code corrected:

Kasu has removed this redundancy.

6.12 Typos

Informational | Version 1 | Code Corrected

CS-KASU-Core-015

We found a few typos in the natspec. Following is a non-exhaustive list:

1. `emitUserLoyaltyRewardBatch`: for `ksuTokenPrice` is mentioned **Iz** zero, the current price
2. During the third step of the clearing, in else branch is written "set the **cleating** configuration"
3. `requestWithdrawal`: "If the **deposit** is done during". Deposit should be withdraw.

Code corrected:

Kasu has resolved the above mentioned typos.

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 Boolean Storage Variables

Informational **Version 1** **Acknowledged**

CS-KASU-Core-001

The default EVM storage type is `uint256`. All other types usually get masked. This also happens for boolean values. Hence, it is more efficient to use `uint256` values instead of boolean values.

Examples are `allowList`, `blockList`, `_exchangeAllowlist`, `_isUser` and flags like `_userCanOnlyDepositToJuniorTrancheWhenHeHasRKSU`.

Acknowledged:

Kasu has acknowledged this issue and replied that the system is deployed on a L2. Thus, the gas savings are not relevant. Additionally, boolean values improve the code's readability.

7.2 Gas Optimizations

Informational **Version 1** **Acknowledged**

CS-KASU-Core-002

- `LendingPool.updateTrancheInterestRate()` iterates over `_futureTrancheInterests` of a given tranche an pops all the interest rates in the future as well as the current epoch and overwrites the interest rates of the current epoch with a new value. Therefore, in

```
if (_futureTrancheInterests[tranche][i].epoch >= applicableEpoch)
```

>= can be replaced with >.

- Some loops write to storage in each iteration. This could be avoided by writing once when the loop has finished. E.g.,

  - The actions that pop or delete from state arrays or state mappings like `allUsers.pop()` or `_trancheInterestIndex[tranche]`.
  - The actions that sum certain state variables up like: `_applyTrancheInterest` writes to `feesOwedAmount += feesAmount` in each iteration, `_emitUserLoyaltyReward` writes to `totalUnclaimedRewards += ksuReward` and `userOwedAmount += userInterestAmount` in each iteration,

  `_decreasePendingDepositAmount` writes to `totalPendingDepositAmount -= amount` in each iteration

- `KSULocking.enableFeesForUser()` calls `_updateUserRewards()` if the fee recipient has not been enabled for the user, meaning `_userRKSUBalanceForFees()` returns 0 for the user. Hence, `_updateUserRewards()` actually does not update the `_rewards` for the user.

- `UserLoyaltyRewards.MAX_REWARD_EPOCH_RATE` can be defined as constant. This does not increase gas costs but is a bit misleading.

Version 6

- Solidity does not tightly pack structs or state variables by default. Therefore, manual packing is required and might make sense to use in some situations. In most cases structs are already optimized but Kasu might want to review if some cases e.g., `FixedTermDepositsEpoch` as the contained values are usually read together.
- In some cases the same storage variable is read or written multiple times. Kasu might consider caching these variables in memory before applying the change to storage. For example:
    - `fixedTermDepositConfig.fixedTermDepositStatus` is read twice in `verifyFixedTermDepositParameters()`.
    - `lendingPoolFixedTermDepositConfigCount[lendingPool]` is read twice in `addLendingPoolTrancheFixedTermDeposit()`.
    - `endFixedTermDeposit()` reads from storage `_lendingPoolFixedTermDepositIds[lendingPool].length` at each iteration of the loop.
    - `_lockFixTermDeposit()` reads `_lendingPoolFixedTermDepositNextId[lendingPool]` from storage twice.
    - `_endFixedTermDeposit()` reads both `depositIds[i]` and `depositIds.length` twice from storage twice.
    - `_fixedTermDepositsClearingPerEpoch()` reads twice `applyFixedTermInterests` from storage.
    - `applyFixedTermInterests()` reads `_fixedTermDepositsClearingPerEpoch[lendingPool][targetEpoch].status` multiple time from storage.
    -
- In several occurrence, mapping value access or array indexing is performed multiple time where it could be cached, for example:
    - In `updateLendingPoolTrancheFixedInterestStatus()`, the location of `_lendingPoolFixedTermDepositConfigurations[lendingPool][fixedTermConfigId]` is computed twice where it could have been cached.
    - In `_initializeLendingPoolFixedTermDepositsProcessing()`, the location of `_fixedTermDepositsClearingPerEpoch[lendingPool][targetEpoch]` could be cached.
    - In `applyFixedTermInterests()`, the location of `_fixedTermDepositsClearingPerEpoch[lendingPool][targetEpoch]` could be cached.
    - In `endFixedTermDeposit()`, the location of `_lendingPoolFixedTermDepositIds[lendingPool]` could be cached.
    - In `updateFixedTermDepositAllowlist()`, the location of `fixedTermDepositsAllowlist[lendingPool][configId]` could be cached.

Code partially corrected and acknowledged:

Kasu applied resolved the following optimizations:

- `updateTrancheInterestRate()` using >=,
- `updateLendingPoolTrancheFixedInterestStatus` value caching
- `endFixedTermDeposit` value caching
- `_initializeLendingPoolFixedTermDepositsProcessing` value caching
- `_endFixedTermDeposit` value caching `depositIds[i]`

Kasu acknowledged the remaining optimizations as the project is supposed to be deployed on a L2 and gas costs are not as relevant as on L1.

7.3 Inefficient Loops

Informational **Version 1** **Acknowledged**

CS-KASU-Core-003

In some cases the system loops over many users. E.g., to distribute the loyalty rewards in `emitUserLoyaltyRewardBatch`. The function will compute the reward for each user and store it. Users can then claim their rewards. These operations are gas expensive and might be better implemented with e.g., Merkle proofs. This also applies to the loss claims which might be done more efficiently by e.g., Merkle proofs.

Acknowledged:

Kasu replied that it was a requirement to perform all calculations on-chain and the gas consumption is not a priority as the project will be deployed on a L2.

7.4 Reentrancy Protection

Informational **Version 1** **Acknowledged**

CS-KASU-Core-004

When a user deposits and additionally uses the in-build swap function, there are multiple reentrancy possibilities. Even though we could not identify any issue, Kasu might consider adding reentrancy protection to obvious reentrancy possibilities.

Acknowledged:

Kasu also cannot identify any issue and decided to keep the code unchanged and replied:

> "In the case of the deposit before the swap, all the external calls are done before or after the state-changing code."

7.5 Storage Packing

Informational **Version 1** **Acknowledged**

CS-KASU-Core-005

Currently, there is potential to optimize the storage slots used by (1) using smaller types for variables that do not need the size of a unit256 (like `clearingPeriodLength`) and (2) re-organizing the storage slots

to pack them tightly. However, this adds little gas costs to unpack the variables compared to an SLOAD. Hence, it makes most sense, when two values that are always used together are packed into one slot.

Additionally, structs can be optimized to use less storage in the same way (e.g., `initialEpochStartTimestamp`).

Acknowledged:

Kasu has acknowledged this possibility with:

> "The system will be deployed on an L2. The increase in gas costs is not significant enough to decrease readability."

7.6 Technical Unnecessary Intermediate Clearing Steps

Informational **Version 1** **Acknowledged**

CS-KASU-Core-006

When `doClearing()` is called, the `ClearingStatus` is evaluated and set. Some loops in the clearing process may be too big to fit in one block. Hence, it is possible to use batch operations in these scenarios and split the transaction into multiple transactions if needed. This is possible when `ClearingStatus` is set to `STEP2_PENDING` or `STEP4_PENDING`. However, the other steps are executed atomically with no interruption possible. These steps are purely cosmetic and technically not needed.

Acknowledged:

Kasu replied with:

> "Makes to code more readable and easier extendable in the future. The system will be deployed on an L2. The increase in gas costs is not significant enough to remove it."

7.7 Unused Imports

Informational **Version 1** **Code Partially Corrected**

CS-KASU-Core-008

The following contracts import code that is not needed:

- `KasuAccessControllable` imports `CommonErrors`
- `IPendingPool` imports `IAcceptedRequestsExecution`
- `AcceptedRequestsExecution` imports `CommonErrors`, `ILendingPoolTranche` and `ILendingPoolManager`
- `IClearingCoordinator` imports `IPendingRequestsPriorityCalculation`
- `PendingRequestsPriorityCalculation` imports `CommonErrors`
- `LendingPoolFactory` imports `TransparentUpgradeableProxy` and `IERC20`
- `PendingPool` imports `ILendingPoolTranche` and `ILendingPoolErrors`
- `LendingPoolTranche` imports `ERC1155Upgradeable` and `ILendingPoolErrors`
- `UserManager` imports `ILendingPoolTranche`

Version 6

- `FixedTermDeposit` imports `CommonErrors`

Code corrected:

Kasu has removed all the unused imports.

7.8 Unused Return Value

Informational Version 6

CS-KASU-Core-021

In `FixedTermDeposit`, the return value of `IPendingPool.requestPriorityWithdrawal()` called in `applyFixedTermInterests()` is not used.

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 Bypassing Waiting Time

Note Version 6

In case the waiting time to withdraw is bigger than the lowest lock time, a user could convert their shares into a fix term deposit and create a withdrawal request for the fix term position. When the fix term position expires the withdrawal will be processed immediately. This might be faster than waiting for the usual priority based withdrawal queue.

8.2 Deposit and Withdrawal Request Can Be Submitted in a Clearing Period

Note Version 1

While the period is in the clearing phase but the clearing execution is not pending, a new deposit or withdrawal request can be submitted. This might be before clearing execution started or after clearing ended. The request will go into the next period. Consequently, only while the clearing execution is pending no new cancellation requests can be submitted.

8.3 Request Processing

Note Version 1

If the clearing manager misses to call `doClearing` in an epoch, `doClearing` will be called in a later epoch. When this happens, the request from earlier epochs will be processed all together in the latest epoch, (when `doClearing` is called).

8.4 Violation of Max Tranche Interest

Note Version 6

It is enforced that fixed interest terms can be at maximum the maximal interest rate of their corresponding tranche. But the maximum interest rate of a tranche can be changed in `setMaxTrancheInterestRate()` by the admin. If the new max. tranche interest rate is below the fixed term rate from before, this would break the invariant and the fixed term interest rate would exceed the max. rate of a tranche.