PUBLIC

Code Assessment

of the Kuma Protocol Smart Contracts

October 4, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

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

The KUMA protocol is designed to tokenize KUMABond NFTs into KIB ERC-20 tokens. Interests are distributed through the rebasing mechanism of the token. ERC-20 ist the most common token standard and hence these KIB tokens are compatible with various decentralized finance protocols. The system has safeguards such as a Deprecation Mode to allow for a graceful shutdown and uses UUPS proxy pattern for upgradability.

The most critical subjects covered in our audit are asset solvency and functional correctness. This includes the yield distribution for the rebasing token.

The general subjects covered are the documentation, integrability into the DeFi ecosystem and efficiency.

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	2
• Code Corrected	1
• Specification Changed	1
Low-Severity Findings	3
• Code Corrected	3

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 Kuma Protocol 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	24 August 2023	093dacdd8457ff4edd697f59341e1b77b5e93d63	Initial Version
2	2 October 2023	2f3fc3d3fff4f0e5b9bc003ef0e9d3d26b70bd5f	After Intermediate Report
3	4 October 2023	a005be2cb13b2b1a6ac054e7ba65ce255b5adc5a	Updated README

For the solidity smart contracts, the compiler version `0.8.17` was chosen. For the purposes of this review we assume the smart contracts will be deployed on Ethereum and executed in the Ethereum EVM Version London. For deployment on other chains/L2s the EVM compatibility must be ensured.

The following files in the folder `src` were in the scope of this review:

- KBCToken.sol
- KIBToken.sol
- KUMAAccessController.sol
- KUMAAddressProvider.sol
- KUMAFeeCollector.sol
- KUMASwap.sol
- MCAGRateFeed.sol

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope. This review focuses on the implementation of the smart contracts, the economic model / incentives for and the behavior of the individual actors has not been investigated in depth.

2.2 System Overview

The KUMA protocol allows to tokenize KUMABond NFT tokens into KIB ERC-20 tokens. KUMA Bond NFTs are backed by real world bonds and are issued by Mimo labs, a regulated entity. KIB ERC-20 tokens are rebasing, the balance of the token holders increases once per epoch as the yield of the bonds is accrued. The yield distributed is limited by the bond with the minimum coupon (interest of the bond) the protocol holds and the current central bank rate for this bond type.

Holders of KUMABond NFT tokens can sell their NFT to the protocol in exchange for KIB ERC-20 tokens. These ERC-20 can then be used across the decentralized finance protocols. Anyone accruing sufficient KIB tokens can buy the bond NFT back from the KUMA protocol.

The system consists of the following contracts:

Infrastructure contracts, deployed once for the protocol:

KUMAAddressProvider:

Central address provider for the KUMA protocol. Used by the various protocol contracts to fetch contract addresses. Addresses of the system contracts can be added/updated by the `KUMA_MANAGER_ROLE`. The implementation of the contract is upgradable.

KUMAAccessController:

Central contract for the role-based access control within the system. Allows the `DEFAULT_ADMIN_ROLE` to grant/revoke roles to addresses. Several privileged roles exist as described in the trust model. This is the only contract that is not upgradable.

MCAGRateFeed:

Pricefeed used by the KUMASwap and KIBContracts to query the reference central bank rate. Underlying oracle is assumed to be the MCAGAggregator for the specific risk category. This rate feed enforces a minimum coupon rate and ensures the freshness of the data. The implementation would support conversion of the decimals if needed, however since both the value received by the oracle and the result are in 27 decimals, this functionality is unused. The implementation of the contract is upgradable.

For each supported risk category of bonds (defined by the bond's currency, issuer and term) a deployment of the following contracts exists:

KUMASwap:

Core contract of the protocol. Handles the exchange of KUMABond NFTs for KIB tokens. When KUMABond NFTs are sold to the protocol, the corresponding value of the bond determines the quantity of KIB tokens that are minted in return. All accounting is based on epochs, when buying/selling bonds the valuation used is the bond's value at the end of the last epoch.

Yield distributed to the KIB token is limited by the bond with the lowest coupon held by the KUMASwap contract or, if lower, the current central bank rate.

During normal operation unprivileged users can execute three state changing functions:

- `sellBond()`: KUMABond NFT holders can sell their token to the KUMASwap contract of the corresponding risk category of their Bond. The amount of KIB tokens minted in exchange is the bond value, a part thereof is minted to the KUMAFeeCollector to pay the protocol fee.
- `buyBond()`: KIBToken holders can buy bonds held by the contract. If expired bonds are held by the contract, these must be bought first. While the bond has been held over time at the KUMASwap contract, it has realized a certain profit in interests. At the same time a part of this, potentially limited by the bond with the min coupon / the central bank rate has been distributed as yield to the KIB token holders. The amount of KIB tokens to be paid is determined based on the realized value (bond value + what has been distributed to the KIB token holders). Only if this is equal or more compared to the value of the bond, the actual KUMABondNFT is transferred to the buyer. If the KUMABondNFT has accrued more interest than distributed to the KIB token, a Clone NFT with the following parameters is minted to the buyer:

```
IKBCToken.CloneBond({
            parentId: tokenId,
            issuance: previousEpochTimestamp,
            coupon: yield,
            principal: realizedBondValue
        })
```

where the `tokenId` is used in a mapping to retrieve the information about the underlying KUMABond NFT, the coupon is set to the current yield of this KUMASwap instance and the principal is set to the amount of KIBT the bond had backed. This Clone bond has to be redeemed separately with representatives of the KUMA protocol. Once KUMA gets hold of this CloneBond, a privileged role can redeem it and claim the actual KUMABondNFT from the KUMASwap contract.

Both functions to buy/sell bonds are only active when the contract is not paused.

- `expireBond()`: Checks if the given NFT id is held by the contract and is indeed expired. If yes, adds it to the expired bonds and calls `refreshYield` on the KIB token to halt yield accrual.

Deprecation Mode:

The protocol requires users to buy the bond NFTs back eventually. The amount of KIB token in circulation is exactly the amount required to buy back all bonds of the protocol. Hence, should some KIB tokens become inaccessible or in case any single party is unable to acquire sufficient KIB tokens to buy back bonds the system gets stuck.

Deprecation Mode allows to gracefully shutdown the KUMA Swap instance: A privileged role can buy back the bonds for a predefined stablecoin instead of KIB tokens. KIB holders can then redeem their KIB token proportionally for their share of the stablecoin.

Initiation of the deprecation mode is a two step process with a time delay: `KUMA_MANAGER_ROLE` calls `initializeDeprecationMode()`, after a hardcoded delay of 2 days again only the `KUMA_MANAGER_ROLE` can enable the deprecation mode. `uninitializeDeprecationMode()` allows the same role to abort the initialization of the deprecation mode.

Several functions for privileged roles to configure the contract and multiple view function to query the contracts state exist.

The implementation of the contract is upgradable.

KIBToken:

Rebasing ERC20 token with 18 decimals. Issued by the KUMASwap contract to tokenize the KUMABondNFT. The rebasing is according to the yield accrual, either it is halted or it increases the balances. The rate is limited by the bond with the minimum coupon held by the KUMASwap contract or, if lower, the current central bank rate.

KIB stands for KUMA Interest Bearing Token and is the technical name of the token. For actual deployments the token name will be made of the currency and term, e.g. USK (one year US treasury bonds in USD).

Rebasing happens once per epoch. Function `refreshYield` allows anyone/keepers to trigger this permissionlessly. The protocol relies on the keepers to call this function once per epoch. Whenever needed (before balances of accounts change, e.g. upon transfer/mint/burn), the implementation ensures to refresh the yield first. Interests are distributed as cumulative yield, not linearly.

In addition to the default ERC20 functioanlity this token implements ERC20Permit.

The implementation of the contract is upgradable.

KBCToken:

KUMACloneBond Token. Issued to the buyers of KUMABondTokens by the KUMASwap contract instead of them receiving the actual KUMABond token in case the realized value (= value distributed as yield to the KIB token) during the time the KUMASwap held this NFT bond is less than what the Bond actually accrued.

A clone bond is always paired with a parent bond from the reserve and will have a lower coupon overriding its parent coupon when the bond is valued. For each clone bond present, the parent bond cannot be redeemed by a user and can only be redeemed by the MCAG multisig when the clone bond has been redeemed by MCAG.

Once redeemed / once the `KUMA_SWAP_CLAIM_ROLE` received the token, this role can redeem it on KUMASwap and claim back the original NFT.

This token is a standrd ERC721 NFT token and is freely transferable.

The implementation of the contract is upgradable.

KUMAFeeCollector:

Fee collector for the KUMA protocol. One instance will be deployed per risk category / KIB token. Fees are taken when a user sells a bond to the KUMASwap contract. A part of the newly minted KIB tokens are minted to the fee collector.

The FeeCollector contract maintains a list of `_payees`. For each payee there is a certain number of `_shares` to determine his proportion of the total fee collected upon fee distribution.

Function `release()` allows anyone to trigger the distribution if a non-zero balance is available and there are `_payees` to receive the tokens.

The KUMAManager can add/remove or update the shares of payees either through the individual functions `addPayee`, `removePayee`, `updatePayeeShare` or the function `changePayees` which allows the manager to directly pass an array with the new payees and new shares.

The implementation of the contract is upgradable.

Upgradability:

Several contracts of the KUMA protocol are upgradable, for upgradability the UUPS proxy pattern is used: A minimal proxy is deployed which delegatecalls into the implementation, the upgradability functionality is handled by the implementation contract.

The reviewed contracts are the second version of the protocol: They can be used for both, to upgrade the previous version of existing deployments or to directly deploy new instances.

2.2.1 Trust Model & Roles

Untrusted roles:

- KUMABond NFT sellers/buyers
- KIB token holders
- Permissionless keepers

The protocol relies on keepers especially to halt yield distribution in case expired bonds exist within the system. Furthermore, keepers must ensure yield is calculated & updated once per epoch, otherwise yield is not distributed for this epoch.

Trusted roles:

- `DEFAULT_ADMIN_ROLE`: Fully trusted. Manages the roles: grants / revokes roles to addresses. If compromised, the whole system is compromised.
- `KUMA_MANAGER_ROLE`: Main administrative role. Has the privileges to:
- Upgrade upgradable contracts
- Update contract addresses in KUMAAddressProvider
- Administrate the KUMAFeeCollector and MCAGRateFeed

In KUMASwap:

- Set the parameters: Fees, MaxCouponTolerance, MaxCouponTolerance, DeprecationStableCoin, initialize / uninitialize deprecation mode, enableDeprecation mode after the timeout
- When the contract is in deprecation mode: Calling `buyBondForStableCoin()`, trusted to pay the proper amount in stablecoin for the bond.

`KUMA_MINT_ROLE`: Can mint KIB tokens, role should be held by the respective KUMASwap instance. `KUMA_BURN_ROLE` Can burn KIB tokens, role should be held by the respective KUMASwap instance.

Furthermore the following roles exist to update the respective parameters:

- KUMA_SET_EPOCH_LENGTH_ROLE
- KUMA_SWAP_CLAIM_ROLE
- KUMA_SWAP_PAUSE_ROLE
- KUMA_SWAP_UNPAUSE_ROLE
- KUMA_SET_URI_ROLE

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	0
Low-Severity Findings	0

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	2

- Wrong Formula in Documentation Specification Changed
- getRate() Blocks the System in Case of Stale Data Code Corrected

Low-Severity Findings	3

- Initialize Implementation Contract Code Corrected
- Setting _lastRefresh to the Next Epoch Initially Code Corrected
- Unused Event Code Corrected

6.1 Wrong Formula in Documentation

Correctness Medium Version 1 Specification Changed

CS-MKP-008

In the documentation of the interest bearing logic the formula for the calculation of the `previousEpochCumulativeYield` is incorrect:

```
previousEpochCumulativeYield is calculated as follows:

newPreviousEpochCumulativeYield=oldPreviousEpochCumulativeYield*(1+yield)^
timeElapsedToEpoch

Here timeElapsedToEpoch refers to the time elapsed between the last cumulativeYield
refresh and the previousEpochTimestamp.
```

It's not `oldPreviousEpochCumulativeYield` but `_cumulativeYield`, the value calculated at the last cumulative yield refresh. While the last refresh could be at the last epoch timestamp, this generally isn't the case.

Furthermore the in the Docs, yield is calculated as

$$yield = 1 + annualRate^{\frac{1}{31536000}} \times 10^{27}$$

On the same page, the following is mentioned:

$$newCumulativeYield = oldCumulativeYield \times (1 + yield)^{elapsedTime}$$

Hence, it is not totally clear for the reader if this yield and the variable `_yield` in the codebase is the exponential yield or 1 + exponential yield.

Specification changed:

The formula for `newPreviousEpochCumulativeYield` has been corrected, `oldPreviousEpochCumulativeYield` has been replaced with `oldCumulativeYield`.

In the formulas for the calculation of `newCumulativeYield` and `newPreviousEpochCumulativeYield`, `(1 + yield)` has been corrected to `yield`.

6.2 `getRate()` Blocks the System in Case of Stale Data

Design | Medium | Version 1 | Code Corrected

CS-MKP-007

In `MCAGRateFeed.getRate()`, if for any reasons, the queried oracle fails to return a fresh, non-stale response, this function reverts. Consequently, all functions which eventually call into `getRate()` will fail as well. This includes any action where the yield is refreshed, including minting, burning and transfers of the KIB tokens.

Most notably this inhibits the following functions (incomplete list):

1. `KUMASwap.expireBond()` - As long as the oracle data is stale it will not be possible to mark bonds as expired.
2. `KUMASwap.enableDeprecationMode()` - Deprecation mode cannot be enabled.
3. `KIBToken.setEpochLength()` - Epoch length cannot be changed.

Code corrected:

Mimo Initiative lt addresses the issue in `KIBToken._refreshYield` through a `try-catch` statement. Should the rate feed fail for any reason the rate used is the current minimum coupon.

This resolves the issue for all functions that query the rate through `_refreshYield()`. Besides them, direct queries of the rate occur in `initialize()` and `sellBond()`. Successful calls to these functions are prevented if the call to the rate feed fails.

6.3 Initialize Implementation Contract

Design | Low | Version 1 | Code Corrected

CS-MKP-006

The first caller can execute `KUMASwap.initialize()` directly on the implementation contract. The protection intended to prevent this is ineffective. Note that this has no actual implications since this implementation contract is intended to be used via the Proxy.

The constructor of KUMASwap looks as follows:

```
constructor() initializer {}
```

The `initializer()` modifier is called to set `_initialized` to 1. It appears that this is done to prevent execution of the `initialize` function on the implementation contract which is protected by the `initializer` modifier.

In this second iteration of KUMASwap the `initialize` function is protected by another modifier: `reinitiliazer(2)`.

Since during deployment `_initialized` is set to 1 in the storage of the implementation, the first caller can pass the modifier `reinitializer(2)`:

```
modifier reinitializer(uint8 version) {
    require(!_initializing && _initialized < version, "Initializable: contract is already initialized");
    _initialized = version;
    _initializing = true;
    _;
    _initializing = false;
    emit Initialized(version);
}
```

OpenZeppelin offers a function to disable initializer: `_disableInitializers()` and recommends to not leave an implementation contract uninitialized.

```
It is recommended to use this to lock implementation contracts that are designed to be called through proxies.
```

See: https://docs.openzeppelin.com/contracts/4.x/api/proxy#Initializable

---

Code corrected:

In the constructor of every implementation contract, `_disableInitializers()` is invoked to lock the (re)initialize function.

6.4 Setting `_lastRefresh` to the Next Epoch Initially

Design | Low | Version 1 | Code Corrected

CS-MKP-005

When `KIBToken` gets initialized, if `block.timestamp` is not divisible through `epochLength`, `_lastRefresh` is set to the next epoch.

`KIBToken.mint()`, which is executed when a bond is sold to the protocol would revert in case of `block.timestamp < _lastRefresh`.

The intention of this is unclear. Anyone can advance `_lastRefresh` to the current `block.timestamp` by calling `KIBToken.refreshYield()`.

Code corrected:

Mimo Initiative lt changed `KIBToken.initialize()` to set `_lastRefresh` to `block.timestamp`.

6.5 Unused Event

Design | Low | Version 1 | Code Corrected

CS-MKP-004

The event `IKUMASwap.MaxCouponsSet` is defined but not used. In the current implementation, `MAX_COUPONS` is defined as a constant.

Code corrected:

This event has been removed.

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 Define Relevant Fields in the Events as Indexed

Informational **Version 1**

CS-MKP-001

Although some event fields are already defined as indexed, it might make sense to define further ones as indexed as well. Defining event fields as indexed makes searching for specific addresses/values easier.

7.2 Epochs to Avoid Dust

Informational **Version 1**

CS-MKP-002

The documentation states:

```
Avoid leftover residual dust amounts when transferring all of a user's balance,
since most frontend wallets do not refresh a user's balance on a per second basis.
```

Working with epochs helps to reduce this from happening but it still happens when a user crafts a transaction before an epoch has ended but this transaction is only included in a block in the next epoch.

7.3 Gas Optimizations

Informational **Version 1**

CS-MKP-003

Gas consumption may be reduced in many different parts of the code. Following is a non-exhaustive list of possible gas optimizations:

- Corresponding `KIBToken` and `KUMABondToken` could be stored locally in KUMASwap, rather than fetching them through 2 external calls every time
- `KIBToken._getPreviousEpochTimestamp()` can be simplified to return `(block.timestamp/epochLength)*epochLength`
- In `KIBToken._calculateCumulativeYield()` and `KIBToken._calculatePreviousEpochCumulativeYield()`, by changing the strict less-comparison (<) to less-or-equal and simplifying the rest of the function
- The if-else-statement on `_coupons.length()` in `KUMASwap.sellBond()` can be reordered, as during the execution life-cycle of the smart contract, the possibility of having no coupons in the reserve list is low
- Adding an existing element to a `Set` does not fail but returns a boolean. Therefore, it is not necessary to check whether the element is already present in the set

- In `KUMAFeeCollector.changePayees()`: `_totalShares` after clearing the already existing payees, is zero. Therefore, `KUMAFeeCollector.changePayees()` does not need to set the local variable `totalShares` to zero explicitly

The following functions are executed once or rarely, nevertheless:

- Parameter `newUri` in `KBCToken.setUri()` could be defined as calldata, similarly `name` and `symbol` in `KIBToken.initialize()`
- `KIBToken.initialize()` the first if-statement (check on `epochLenght`) can be removed, as it is a subset of the second if-statement

Not a gas optimization but code duplication could be avoided:

- Calling `_updateMinCoupon()` in `KUMASwap.buyBond()` can be moved out of if-else to avoid duplication of code

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 Bond Default

Note Version 1

If a bond defaults or is expected to default, it may start to trade at a sharply reduced price. The KIB tokens are still backed by the bonds which now have a much lower value than anticipated. Hence the valuation of the KIB tokens will drop accordingly. `buyBond()` / `sellBond()` continue to buy/sell KUMABond NFTs for the usual exchange rate - except that the actual value of a KIB token is now much lower. It is unclear if the market price of the KIB tokens reflect this or whether there is an arbitrage opportunity.

If yield accrual is not halted, KIB tokens will devaluate further.

When redeeming such a KUMABond NFT through MCAG the underlying bond will be sold on the secondary market.

Bonds of the same risk category can have a different expiry dates as they could be issued at different times. Some bonds of a risk category might expire in 1 month, some in 6 months. If the default of an issuer is foreseeable or expected within prolonged timeframe, KUMASwap might end up with the bonds expiring far in the future, while all bonds that expires soon have been bought from the protocol.

In rare cases, a bond might be restructured to avoid a default. The current smart contracts do not support this.

8.2 Bridges and Rebasing Tokens

Note Version 1

The documentation mentions the KIB tokens are intended to be bridged to other chains: https://docs.kuma.bond/kuma-protocol/kuma-protocol/defi-integrations/bridges

While technically possible, bridging rebasing tokens is tricky to be done correctly and requires extra care. Even with only dust amounts of KIB tokens remaining locked at the bridge, eventually the deprecation mode will have to be used since insufficient KIB tokens to buy back all the bonds will be available.

8.3 CloneBond Token Does Not Respect Blacklist

Note Version 1

Contrary to the KUMABondNFT, the CloneBondToken does not respect the blacklist. The following interesting behavior can be observed: If a blacklisted address calls `buyBond()` it succeeds when a clone bond is issued but fails if the KUMABondNFT is transferred.

Mimo Initiative lt states this behavior is intended and blacklisted users won't be able to redeem with Mimo Capital.

8.4 Deprecation Mode Possibility

Note Version 1

As mentioned in the Docs:

> *KIBT is minted and burned such that the total supply is always sufficient to buy out all of the KUMA Bonds NFTs held in the contract. However, in extreme cases, some of the KIBT supply may become inaccessible (e.g. if large amounts of KIBT are hacked or lost to unknown addresses).*

It is worth mentioning that the likelihood of requiring Deprecation Mode is likely higher than what is outlined in the documentation. To buy a bond of the protocol one party must accrue sufficient KIB tokens. This may prove to be hard in practice, even when ignoring that some of the KIBT supply may be inaccessible. Especially, when the liquidity of the token is low compared to the bond value to be bought, acquiring tokens from pools like Uniswap might be prohibitively expensive as their price increases especially if large quantities are bought.

8.5 KIB Price Determination & Volatility

Note Version 1

It may be obvious that one KIB token should be priced at one unit of the underlying principal token. However due to low liquidity, constrains around redemption of KIB tokens (accruing sufficient KIB tokens, buying the bond of the protocol, off-chain redemption) the price may fluctuate. Furthermore, low liquidity in e.g. Uniswap pools result in notable price volatility / slippage when trading as well as potential price manipulation.

8.6 KUMAAddresProvider: KUMAAccessController Not Immutable

Note Version 1

The KUMAAccessController contract is the sole contract that does not implement upgradability features. In the KUMAAddressProvider, the KUMAAccessControler is the only address that cannot be updated after it has been initialized.

Note that since the KUMAAddressProvider is upgradable, the implementation may change and allow to update the address of the KUMAAccessController.

8.7 Risks of Low Liquidity

Note Version 1

Low liquidity of the KIB tokens may have negative effects when using the token in DeFi systems such as Uniswap, Balancer, Curve and more. This includes price volatility/manipulation, slippage and arbitrage opportunities.

Furthermore low liquidity may have negative consequences when attempting to raise sufficient liquidity to buy a bond of KUMASwap. Notably this may delay the removal of expired bonds

8.8 Stuck Stablecoins

Note Version 1

Deprecation mode can be enabled if it is no longer possible to acquire sufficient KIB tokens to buy back bonds. In this mode, the `KUMA_MANAGER_ROLE` can buy back the bonds for the defined stablecoin. KIB token holders can then exchange their tokens for a proportional share of the stablecoin held by the contract. By design, if some KIB tokens are no longer accessible they cannot be redeemed for their share of the stablecoins, hence these stablecoins will then be locked at the KUMASwap contract.

8.9 Use as Collateral

Note Version 1

The upgradability of both the KIB token contracts and the contracts that control token minting (KUMASwap) and to a smaller extent the deprecation mode could discourage lending protocols from accepting the token as collateral.

8.10 `_couponTolerance` Protection Mechanism

Note Version 1

When a bond is sold to the protocol, a check ensures that the bond's coupon is not considerably lower than the existing `minCoupon` rate. However, this check is only applicable to the coupon rate of the newly offered bond. Multiple bond sales to the protocol, potentially in quick succession, could reduce the `minCoupon` by a larger margin.

8.11 balanceOf, totalSupply and getUpdatedCumulativeYield Potentially Returning Virtual State

Note Version 1

`balanceOf` and `totalSupply` are both view functions according to the ERC-20 standard, hence can't modify state.

To determine the balance / totalSupply the last `_calculatePreviousEpochCumulativeYield()` has to be evaluated. If `refreshYield()` has already been executed in this epoch, the value is simply read from storage and returned. Otherwise the new value is calculated and returned but not stored.

Whenever these functions are used internally, this happens after an execution of `_refreshCumulativeYield()` and `_refreshYield()` hence the value has already been updated and is just returned from storage.

External callers should be aware that when calling these functions they may get "virtual" balance and the contract state itself has not updated yet. The values will only be updated with the next state changing call to `_refreshCumulativeYield()`.

Similarly, this applies to the external view function `getUpdatedCumulativeYield()` which is called by KUMASwap in `buyBond()` and `sellBond()` before a state changing function (e.g. minting/burning) tokens are called which recalculates and this time stores the updated the value.