KUMA Protocol - Versus contest Findings & Analysis Report

2023-06-23

Table of contents

- Overview
    - About C4
    - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (1)
    - [H-01] TRANSFERING KIBToken TO YOURSELF INCREASES YOUR BALANCE
- Medium Risk Findings (4)
    - [M-01] `KUMABondToken.approve()` should revert if the owner of the tokenId is blacklisted
    - [M-02] `KUMAFeeCollector.changePayees()` executes incorrectly when newPayees contains duplicate items
    - [M-03] Price feed in `MCAGRateFeed#getRate` is not sufficiently validated and can return stale price

- [M-04] KUMASwap incorrectly reverts when when `_maxCoupons` has been reached
- Low Risk and Non-Critical Issues
  - `KBCToken.sol`
  - `KIBToken.sol`
  - `KUMAAddressProvider.sol`
  - `KUMAFeeCollector.sol`
  - `KUMASwap.sol`
  - `Blacklist.sol`
  - `KYCToken.sol`
  - `KUMABondToken.sol`
  - `MCAGAggregator.sol`
  - `WadRayMath.sol`
- Gas Optimizations
  - `KBCToken.sol`
  - `KIBToken.sol`
  - `KUMAAddressProvider.sol`
  - `KUMAFeeCollector.sol`
  - `KUMASwap.sol`
  - `KYCToken.sol`
  - `KUMABondToken.sol`
  - `MCAGRateFeed.sol`
  - `MCAGAggregator.sol`
- Mitigation Review
  - Introduction
  - Overview of Changes

    - Mitigation Review Scope
    - Mitigation Review Summary
    - Non-fixed terms will cause variable length bonds with the same nominal yield to have different coupons
- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the KUMA Protocol smart contract system written in Solidity. The audit contest took place between February 17—February 22 2023.

Following the C4 audit contest, 3 wardens (0xsomeone, 0x52, and hihen) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

In Code4rena's Versus contests, the competition is limited to a small group of wardens; for this contest, 5 wardens contributed reports:

1. 0x52
2. 0xsomeone
3. AkshaySrivastav
4. bin2chen
5. hihen

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 KUMA Protocol contest repository, and is composed of 12 smart contracts and 10 interfaces written in the Solidity programming language and includes 1,655 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] TRANSFERING KIBToken TO YOURSELF INCREASES YOUR BALANCE

Submitted by bin2chen, *also found by* AkshaySrivastav, hihen, *and* 0xsomeone

Using temporary variables to update balances is a dangerous construction.

If transferred to yourself, it will cause your balance to increase, thus growing the token balance infinitely.

Proof of Concept

KIBToken overrides `\_transfer()` to perform the transfer of the token, the code is as follows:

```
    function _transfer(address from, address to, uint256 amount)
        if (from == address(0)) {
            revert Errors.ERC20_TRANSFER_FROM_THE_ZERO_ADDRESS()
        }
        if (to == address(0)) {
            revert Errors.ERC20_TRANSER_TO_THE_ZERO_ADDRESS();
        }
        _refreshCumulativeYield();
        _refreshYield();

        uint256 startingFromBalance = this.balanceOf(from);
        if (startingFromBalance < amount) {
            revert Errors.ERC20_TRANSFER_AMOUNT_EXCEEDS_BALANCE
        }
        uint256 newFromBalance = startingFromBalance - amount;
        uint256 newToBalance = this.balanceOf(to) + amount;

        uint256 previousEpochCumulativeYield_ = _previousEpochCu
        uint256 newFromBaseBalance = WadRayMath.wadToRay(newFrom
        uint256 newToBaseBalance = WadRayMath.wadToRay(newToBala

        if (amount > 0) {
            _totalBaseSupply -= (_baseBalances[from] - newFromBa
            _totalBaseSupply += (newToBaseBalance - _baseBalance
            _baseBalances[from] = newFromBaseBalance;
            _baseBalances[to] = newToBaseBalance;//<--------if f
        }
```

```
            emit Transfer(from, to, amount);
        }
```

From the code above we can see that using temporary variables “newToBaseBalance” to update balances.

Using temporary variables is a dangerous construction.

If the from and to are the same, the balance[to] update will overwrite the balance[from] update.

To simplify the example:

Suppose: balance[alice]=10 , and execute transferFrom(from=alice,to=alice,5)

Define the temporary variable: temp_variable = balance[alice]=10

So update the steps as follows:

1. `balance[to=alice] = temp_variable - 5 =5`
2. `balance[from=alice] = temp_variable + 5 =15`

After Alice transferred it to herself, the balance was increased by 5.

The test code is as follows:

add to KIBToken.transfer.t.sol

```
        //test from == to
        function test_transfer_same() public {
            _KIBToken.mint(_alice, 10 ether);
            assertEq(_KIBToken.balanceOf(_alice), 10 ether);
            vm.prank(_alice);
            _KIBToken.transfer(_alice, 5 ether);   //<-----alice tra
            assertEq(_KIBToken.balanceOf(_alice), 15 ether); //<----
        }
```

```
forge test --match test_transfer_same -vvv

Running 1 test for test/kuma-protocol/kib-token/KIBToken.transfe
[PASS] test_transfer_same() (gas: 184320)
Test result: ok. 1 passed; 0 failed; finished in 24.67ms
```

Recommended Mitigation Steps

A more general method is to use:

```
balance[to]-=amount
balance[from]+=amount
```

In view of the complexity of the amount calculation, if the code is to be easier to read, it is recommended：

```
    function _transfer(address from, address to, uint256 amount)
        if (from == address(0)) {
            revert Errors.ERC20_TRANSFER_FROM_THE_ZERO_ADDRESS()
        }
        if (to == address(0)) {
            revert Errors.ERC20_TRANSER_TO_THE_ZERO_ADDRESS();
        }
        _refreshCumulativeYield();
        _refreshYield();

+       if (from != to) {
            uint256 startingFromBalance = this.balanceOf(fro
            if (startingFromBalance < amount) {
                revert Errors.ERC20_TRANSFER_AMOUNT_EXCEEDS_
            }
            uint256 newFromBalance = startingFromBalance - a
            uint256 newToBalance = this.balanceOf(to) + amou

            uint256 previousEpochCumulativeYield_ = _previou
            uint256 newFromBaseBalance = WadRayMath.wadToRay
            uint256 newToBaseBalance = WadRayMath.wadToRay(r

            if (amount > 0) {
                _totalBaseSupply -= (_baseBalances[from] - r
                _totalBaseSupply += (newToBaseBalance - _bas
                            _baseBalances[from] = newFromBaseBalance;
                            _baseBalances[to] = newToBaseBalance;
                        }
    +           }
                emit Transfer(from, to, amount);
            }
```

Alex the Entreprenerd (judge) commented:

> The Warden has shown a way to leverage a programming mistake to duplicate an account balances, because this breaks protocol invariants, I agree with High Severity.

m19 (KUMA) confirmed and commented:

> We agree that this is a high risk issue and we intend to fix this.

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/3

> **Status:** Mitigation confirmed with comments. Full details in reports from 0xsomeone, 0x52, and hihen.

Medium Risk Findings (4)

[M-01] `KUMABondToken.approve()` should revert if the owner of the tokenId is blacklisted

Submitted by hihen, also found by 0xsomeone

It is still possible for a blacklisted user's bond token to be approved.

Proof of Concept

KUMABondToken.approve() only checks if `msg.sender` and `to` are not blacklisted. It doesn't check if the owner of the `tokenId` is not blacklisted.

For example, the following scenario allows a blacklisted user's bond token to be approved:

1. User A have a bond token bt1.
2. User A calls `KUMABondToken.setApprovalForAll(B, true)` , and user B can operate on all user A's bond tokens.
3. User A is blacklisted.
4. User B calls `KUMABondToken.approve(C, bt1)` to approve user C to operate on bond token bt1.

Tools Used

VS Code

Recommended Mitigation Steps

`KUMABondToken.approve()` should revert if the owner of the tokenId is blacklisted:

```
diff --git a/src/mcag-contracts/KUMABondToken.sol b/src/mcag-cor
index 569a042..906fe7b 100644
--- a/src/mcag-contracts/KUMABondToken.sol
+++ b/src/mcag-contracts/KUMABondToken.sol
@@ -146,6 +146,7 @@ contract KUMABondToken is ERC721, Pausable,
             whenNotPaused
             notBlacklisted(to)
             notBlacklisted(msg.sender)
+            notBlacklisted(ERC721.ownerOf(tokenId))
         {
             address owner = ERC721.ownerOf(tokenId);
```

Alex the Entreprenerd (judge) commented:

> The Warden has shown an inconsistency in implementation for the blacklist functionality.

> Because a transfer would still be broken, due to `transferFrom` performing a check on all accounts involved, I agree with Medium Severity.

m19 (KUMA) confirmed and commented:

> We confirmed this issue in a test and intend to fix it:

```
        function test_approve_RevertWhen_TokenOwnerBlacklistedAndApp
            _kumaBondToken.issueBond(_alice, _bond);
            vm.prank(_alice);
            _kumaBondToken.setApprovalForAll(address(this), true);
            _blacklist.blacklist(_alice);
            vm.expectRevert(abi.encodeWithSelector(Errors.BLACKLIST_
            _kumaBondToken.approve(_bob, 1);
        }
```

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/4

> **Status:** Mitigation confirmed by 0xsomeone, 0x52, and hihen.

[M-02] `KUMAFeeCollector.changePayees()` executes incorrectly when newPayees contains duplicate items

Submitted by hihen, also found by 0xsomeone, 0x52, bin2chen, and AkshaySrivastav

When calling KUMAFeeCollector.changePayees() with duplicate payees in `newPayees`, the call is not reverted and the result state will be incorrect.

Proof of Concept

Contract KUMAFeeCollector does not support duplicate payees. The transaction will revert when trying to add duplicate payees in addPayee():

```
    function addPayee(address payee, uint256 share) external overric
        if (_payees.contains(payee)) {
            revert Errors.PAYEE_ALREADY_EXISTS();
        }
        ...
    }
```

But, function KUMAFeeCollector.changePayees() forgets this constraint, which allows duplicate payees to be passed in `newPayees`. This will cause the contract to record an incorrect state and not work properly.

For example, if `newPayees` contains duplicate payee A, all of its shares will be added to `_totalShares`, but `_shares[A]` will only record the last one.

As a result, the sum of all recorded `_shares` will be less than `_totalShares`.

Test code for PoC:

```
diff --git a/test/kuma-protocol/KUMAFeeCollector.t.sol b/test/kı
index f34d9ff..0b3fe46 100644
--- a/test/kuma-protocol/KUMAFeeCollector.t.sol
+++ b/test/kuma-protocol/KUMAFeeCollector.t.sol
@@ -40,6 +40,39 @@ contract KUMAFeeCollectorTest is BaseSetUp {
         );
     }

+    function test_DuplicatePayees() public {
+        address[] memory newPayees = new address[](4);
+        uint256[] memory newShares = new uint256[](4);
+
+        newPayees[0] = vm.addr(10);
+        newPayees[1] = vm.addr(10);
+        newPayees[2] = vm.addr(11);
+        newPayees[3] = vm.addr(12);
+        newShares[0] = 25;
+        newShares[1] = 25;
+        newShares[2] = 25;
+        newShares[3] = 25;
+
+        _KUMAFeeCollector.changePayees(newPayees, newShares);
+
+        // only 3 payees
+        assertEq(_KUMAFeeCollector.getPayees().length, 3);
+        // newPayees[0] and newPayees[1] are identical and both
+        address[] memory payees = _KUMAFeeCollector.getPayees()
+        assertEq(payees[0], newPayees[1]);
+        assertEq(payees[1], newPayees[2]);
+        assertEq(payees[2], newPayees[3]);
+
+        uint256 countedTotalShares = 0;
+         for (uint i; i < payees.length; i++) {
+             countedTotalShares += _KUMAFeeCollector.getShare(pa
+         }
+         // Counted totalShares is 75 (100 - 25)
+         assertEq(countedTotalShares, 75);
+         // Recorded totalShares is 100
+         assertEq(_KUMAFeeCollector.getTotalShares(), 100);
+     }
+
      function test_initialize() public {
          assertEq(address(_KUMAFeeCollector.getKUMAAddressProvic
          assertEq(_KUMAFeeCollector.getRiskCategory(), _RISK_CAT
```

Outputs:

```
forge test -m test_DuplicatePayees
[⠊] Compiling...
No files changed, compilation skipped

Running 1 test for test/kuma-protocol/KUMAFeeCollector.t.sol:KUM
[PASS] test_DuplicatePayees() (gas: 259689)
Test result: ok. 1 passed; 0 failed; finished in 7.39ms
```

Tools Used

VS Code

Recommended Mitigation Steps

KUMAFeeCollector.changePayees() should revert if there are duplicates in `newPayees`:

```
diff --git a/src/kuma-protocol/KUMAFeeCollector.sol b/src/kuma-p
index 402cf71..1a9d86d 100644
--- a/src/kuma-protocol/KUMAFeeCollector.sol
+++ b/src/kuma-protocol/KUMAFeeCollector.sol
@@ -180,7 +180,9 @@ contract KUMAFeeCollector is IKUMAFeeCollect
             }

             address payee = newPayees[i];
-            _payees.add(payee);
```

```
    +               if (!_payees.add(payee)) {
    +                   revert Errors.PAYEE_ALREADY_EXISTS();
    +               }
                    _shares[payee] = newShares[i];
                    _totalShares += newShares[i];
```

m19 (KUMA) confirmed and commented:

> We confirm this issue and intend to fix it.

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to a lack of checks, an inconsistent setting could be achieved, where the total sum of shares would be higher than what would be effectively distributed.

> While the issue can be solved by changing again, because the finding shows a reasonable way to achieve an inconsistent state, which can cause loss of tokens, I agree with Medium Severity.

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/5

> **Status:** Mitigation confirmed by 0xsomeone, 0x52, and hihen.

[M-03] Price feed in `MCAGRateFeed#getRate` is not sufficiently validated and can return stale price

Submitted by 0x52

`MCAGRateFeed#getRate` may return stale data.

Proof of Concept

```
        (, int256 answer,,,) = oracle.latestRoundData();
```

getRate only uses answer but never checks the freshness of the data, which can lead to stale bond pricing data. Stale pricing data can lead to bonds being bought and sold on KUMASwap that otherwise should not be available. This would harm KIBToken holders as KUMASwap may accept bond with too low of a coupon and reduce rewards.

Recommended Mitigation Steps

Validate that `updatedAt` has been updated recently enough:

```
-    (, int256 answer,,,) = oracle.latestRoundData();
+    (, int256 answer,,updatedAt,) = oracle.latestRoundData();


+    if (updatedAt < block.timestamp - MAX_DELAY) {
+        revert();
+    }

     if (answer < 0) {
         return _MIN_RATE_COUPON;
     }
```

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, the system will not check for a stale price.

> In times of high network congestion, liquidations or if the feed has any downtime, the protocol may end-up using a stale price which can leak value.

> For these reasons, I agree with Medium Severity.

m19 (KUMA) disagreed with severity and commented:

> This issue is tricky: central bank rate updates are not expected more than a handful of times a year, at most. We failed to document this properly and the lack of a fresh `updatedAt` check was on purpose.

> If we were only transmitting rates when an actual rate change happens, it's impossible to determine how "fresh" the data should be, a central bank rate

> change could happen every 3 months or twice a year or any timeframe really. So should the staleness check be 90 days, 180 days etc?

> We agree that the oracle transmitter could just repeat the current rate on a daily or weekly basis instead making a stale data check useful again.

> Because central bank rates are not volatile we still disagree with the severity in this case.

Alex the Entreprenerd (judge) commented:

> I empathize with the Sponsor's answer and agree that in a way the maximum delay for the check is hard to determine, however, it's important that we acknowledge that the smart contract cannot "defend itself" unless we offer some sort of constraint that avoids it leaking value incorrectly.

> In contrast to the front-run, which can be viewed as a nuance / technicality;

> The lack of a check would allow the bond to be sold for a time that is not intended, this could also have KumaSwap accept bonds for which the yield is no longer competitive.

> I believe the observation from the sponsor to be valid that the changes may be few and far between, however, because the contract relies on the oracle for its decision making, and no check is there to ensure that the decision "makes sense";

> Given the possibility of:

- Unintended behaviour (selling of bond when that should not be possible)
- Loss of yield as a consequence

> I believe Medium Severity to be the more appropriate of the options.

Alex the Entreprenerd (judge) commented:

> The warden has added a comment in terms of suggested mitigation:

> We agree that the oracle transmitter could just repeat the current rate on a daily or weekly basis instead making a stale data check useful again.

> To quote the sponsor, I believe this would be the best course of action. Even thought the rate does not change frequently, its valuable to have a heartbeat to confirm that all systems are working as intended. Better to experience potential downtime for buys/sells than to have systems unknowingly down during an important rate change.

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/6

> **Status:** Mitigation confirmed by 0xsomeone, 0x52, and hihen.

[M-04] KUMASwap incorrectly reverts when when `_maxCoupons` has been reached

Submitted by 0x52

Selling bonds with coupons that are already accounted will fail unexpectedly.

Proof of Concept

https://github.com/code-423n4/2023-02-kuma/blob/3f3d2269fcb3437a9f00ffdd67b5029487435b95/src/kuma-protocol/KUMASwap.sol#L116-L118

```
        if (_coupons.length() == _maxCoupons) {
            revert Errors.MAX_COUPONS_REACHED();
        }
```

The above lines will cause ALL bonds sales to revert when `_coupons.length` has reached `_maxCoupons` . Since bonds may share the same `coupon` , the swap should continue to accept bonds with a `coupon` that already exist in the `_coupons` set.

Recommended Mitigation Steps

sellBond should only revert if the max length has been reached and bond.coupon doesn't already exist:

```
-   if (_coupons.length() == _maxCoupons) {
+   if (_coupons.length() == _maxCoupons && !_coupons.contains(k
        revert Errors.MAX_COUPONS_REACHED();
    }
```

m19 (KUMA) confirmed and commented:

> Even though this scenario is unlikely to ever happen we have confirmed this issue in a test:

```
    function test_sellBond_WithExistingCouponWhenMaxCouponsReach
        _KUMASwap.sellBond(1);
        IKUMABondToken.Bond memory bond_ = _bond;

        for (uint256 i; i < 364; i++) {
            bond_.coupon = bond_.coupon + 1;
            _KUMABondToken.issueBond(address(this), bond_);
            _KUMASwap.sellBond(i + 2);
        }

        _KUMABondToken.issueBond(address(this), _bond);

        _KUMASwap.sellBond(365);
    }
```

> We intend to fix this.

Alex the Entreprenerd (judge) commented:

> The warden has shown a way in which the system could stop selling coupons due to the handling of duplicate coupons, this relies on a specific condition which may not always happen, for this reason, in lack of an attack that can be used to break the functionality, I agree with Medium Severity.

> m19 (KUMA) mitigated: https://github.com/code-423n4/2023-02-kuma/pull/7

> **Status:** Mitigation confirmed by 0xsomeone, 0x52, and hihen.

Low Risk and Non-Critical Issues

For this contest, 4 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xsomeone** received the top score from the judge.

The following wardens also submitted reports: 0x52, hihen, and bin2chen.

`KBCToken.sol`

KBC-01L: Improper Disable of Initializer (Affected Lines: L30)

The Kuma Protocol repository makes use of the `v4.8.1` dependency of OpenZeppelin with the `Initializer` contract being our focus. The contract contains a dedicated `_disableInitializers` function meant to be invoked by contract `constructor` implementations in contracts that inherit it, ensuring that the contract cannot be initialized **or re-initialized** in the future.

In the current implementation, the `initializer` modifier is simply added to the contract's `constructor` which does not disable the initializer properly as it still permits **a re-initialization to occur**. In the current version of the codebase this does not pose an issue as a `reinitializer` modifier is not in use, however, in a future deployment this may manifest.

`KIBToken.sol`

KIB-01L: Improper Disable of Initializer (Affected Lines: L49)

The Kuma Protocol repository makes use of the `v4.8.1` dependency of OpenZeppelin with the `Initializer` contract being our focus. The contract contains a dedicated `_disableInitializers` function meant to be invoked by contract `constructor` implementations in contracts that inherit it, ensuring that the contract cannot be initialized **or re-initialized** in the future.

In the current implementation, the `initializer` modifier is simply added to the contract's `constructor` which does not disable the initializer properly as it still permits **a re-initialization to occur**. In the current version of the codebase this does not pose an issue as a `reinitializer` modifier is not in use, however, in a future deployment this may manifest.

KIB-02L: Insufficient Initial Epoch Sanitization (Affected Lines: L68)

While the `KIBToken::setEpochLength` function properly evaluates that the `epochLength` provided is not greater-than (>) the `MAX_EPOCH_LENGTH`, no such sanitization is applied during the `KIBToken::initialize` function. We advise the `initialize` function to apply the same logical checks as `setEpochLength`, potentially refactored to an `internal` function that both code segments utilize.

KIB-03L: Discrepant Epoch Inclusivity Definitions (Affected Lines: L79-L81, L342-L345)

The `KIBToken::_getPreviousEpochTimestamp` and `KIBToken::initialize` functions differ in the way they define the "current" and "previous" epoch. In the `initialize` function, the `_lastRefresh` is either set to the current `block.timestamp` or `(block.timestamp / epochLength) * epochLength + epochLength`, while in the `_getPreviousEpochTimestamp` function the value yielded is either `block.timestamp` or `(block.timestamp / epochLength) * epochLength`.

We advise either the assignment in `initialize` to add the `epochLength` to the current `block.timestamp` or the `_getPreviousEpochTimestamp` to subtract the `epochLength` from the `block.timestamp`, the former of which we consider standard.

KIB-04L: Inexistent Enforcement of Minimums / Maximums in Yield (Affected Lines: L331)

The `MAX_YIELD` constant variable declared in `KIBToken` remains unutilized. We advise it to be enforced in the `_refreshYield` function by ensuring that the `lowestYield` being set is at most equal to `MAX_YIELD`, in which case the value of `MAX_YIELD` should be used instead. Additionally, the `MIN_YIELD` value should be enforced in a similar fashion.

`KUMAAddressProvider.sol`

KAP-01L: Improper Disable of Initializer (Affected Lines: L39)

The Kuma Protocol repository makes use of the `v4.8.1` dependency of OpenZeppelin with the `Initializer` contract being our focus. The contract contains a dedicated `_disableInitializers` function meant to be invoked by contract `constructor` implementations in contracts that inherit it, ensuring that the contract cannot be initialized **or re-initialized** in the future.

In the current implementation, the `initializer` modifier is simply added to the contract's `constructor` which does not disable the initializer properly as it still permits **a re-initialization to occur**. In the current version of the codebase this does not pose an issue as a `reinitializer` modifier is not in use, however, in a future deployment this may manifest.

`KUMAFeeCollector.sol`

KFC-01L: Improper Disable of Initializer (Affected Lines: L32)

The Kuma Protocol repository makes use of the `v4.8.1` dependency of OpenZeppelin with the `Initializer` contract being our focus. The contract contains a dedicated `_disableInitializers` function meant to be invoked by contract `constructor` implementations in contracts that inherit it, ensuring that the contract cannot be initialized **or re-initialized** in the future.

In the current implementation, the `initializer` modifier is simply added to the contract's `constructor` which does not disable the initializer properly as it still permits **a re-initialization to occur**. In the current version of the codebase this does not pose an issue as a `reinitializer` modifier is not in use, however, in a future deployment this may manifest.

KFC-02L: Improper Release Event (Affected Lines: L88, L160)

The `KUMAFeeCollector::addPayee` and `KUMAFeeCollector::changePayees` functions may operate on an empty `_payees` data entry. In such a case, if funds to-be distributed are present in the contract the `KUMAFeeCollector::_releaseIfAvailableIncome` function will incorrectly execute `KUMAFeeCollector::_release` which in turn will emit the `FeeReleased` event for the available income. We advise the `_releaseIfAvailableIncome` function to be invoked solely when `_payees.length()` is non-zero in the referenced instances as otherwise incorrect events may be emitted in `addPayee` and `changePayees`.

KFC-03L: Inexistent Duplicate Entry Prevention (Affected Lines: L175-L180)

The `KUMAFeeCollector::changePayees` function does not adequately sanitize the new payees, permitting duplicate entries to exist which will cause the contract to significantly misbehave as it would track the `_totalShares` incorrectly, and perform two payouts with the latest `newShares[i]` value. We advise the code to add a new `if` conditional which causes the code to fail if `_payees.contains(newPayees[i])` evaluates to `true`.

`KUMASwap.sol`

KSP-01L: Inexistent Limitation of Variable Fee (Affected Lines: L360)

The `KUMASwap::setFees` function does not apply any sanitization on the `variableFee` value, permitting a fee to be applied that exceeds the 100% value of the bond (i.e. `PercentageMath::PERCENTAGE_FACTOR`). We advise the `variableFee` input variable of `setFees` to be validated as less than `PERCENTAGE_FACTOR` and preferably less than a stricter value (i.e. 20%) to ensure unfair fees are not applied in the protocol.

This finding also ties in with the slippage-related vulnerability submitted in a separate exhibit whereby a user could submit a transaction with a blockchain state that applies a 5% fee and the fee could change between the transaction’s submission and the transaction’s execution by the network.

KSP-02L: Inexistent Limitation of Bond Fee (Affected Lines: L636-L643)

The `KUMASwap::_calculateFees` function does not apply adequate sanitization to the `fee` that is ultimately applied to the `amount` which can potentially exceed it due to the presence of a `_fixedFee`. We advise the code to mandate that the final `fee` calculated does not exceed the `amount` supplied as an argument and to fail in such a case.

KSP-03L: Improper Disable of Initializer (Affected Lines: L78)

The Kuma Protocol repository makes use of the `v4.8.1` dependency of OpenZeppelin with the `Initializer` contract being our focus. The contract contains a dedicated `_disableInitializers` function meant to be invoked by contract `constructor` implementations in contracts that inherit it, ensuring that the contract cannot be initialized **or re-initialized** in the future.

In the current implementation, the `initializer` modifier is simply added to the contract's `constructor` which does not disable the initializer properly as it still permits **a re-initialization to occur**. In the current version of the codebase this does not pose an issue as a `reinitializer` modifier is not in use, however, in a future deployment this may manifest.

KSP-04L: Unsafe Casting of Term Months (Affected Lines: L100)

The `KUMASwap::initialize` function casts the result of `term / 30 days` to a `uint16` variable unsafely. We advise the casting operation to be performed safely by ensuring that the `term / 30 days` calculation's result does not exceed the maximum value that a `uint16` variable can hold (i.e. `type(uint16).max`). To note, the built-in safe arithmetic of Solidity **does not protect against casting overflows**.

`Blacklist.sol`

BLT-01L: Inexistent Sanitization of State Transitions (Affected Lines: L38, L47)

The `Blacklist::blacklist` and `Blacklist::unBlacklist` functions do not ensure that the previous state of a `_blacklisted[account]` was the opposite of what it is being set to. We advise this sanitization to be applied to prevent misuse of the functions as well as misleading `Blacklisted` / `UnBlacklisted` events.

`KYCToken.sol`

KYC-01L: Weak Definition of Owner (Affected Lines: L49)

The `KYCToken::mint` function accepts a `kycData` argument that is meant to contain data about a KYC'd person. The `kycData` contains an `owner` argument that does not appear to be sanitized, permitting KYC data to be minted to an arbitrary `to` address when the `owner` of the `kycData` may be someone else. We advise the `mint` function to permit minting of the `kycData` by either enforcing `to` to equal the data's `owner` member or by not accepting a `to` argument altogether, minting the `kycData` directly to its `owner`.

`KUMABondToken.sol`

KBT-01L: Potential Approval Blacklist Bypass (Affected Lines: L148)

The `KUMABondToken::approve` function will apply a blacklist check on the `to` as well as `msg.sender` contextual arguments of the call, however, the actual `owner` of the `tokenId` is not validated in contrast to `setApprovalForAll` which disallows an approval to be made by a party that is in the blacklist.

While the approval cannot be actuated on by the `transferFrom` / `safeTransferFrom` functions as they do validate the `from` argument, an approval being made on behalf of a blacklisted owner is an undesirable trait. We advise the code to properly apply the blacklist to the `owner` of the `ERC721` asset, preventing circumvention of the checks if an operator (i.e. `isApprovedForAll`) was created by a blacklisted owner before they were included in the blacklist.

`MCAGAggregator.sol`

MAR-01L: Inexistent Sanitization of Maximum Answer (Affected Lines: L69-L72)

The `MCAGAggregator::setMaxAnswer` function does not apply any sanitization on the input `newMaxAnswer`, permitting even negative values to be set. We advise the function to mandate at least a non-zero `newMaxAnswer`, applying the same check in the `constructor` of the contract.

`WadRayMath.sol`

WRM-01L: Incorrect Code Merge (Affected Lines: L127-L137)

The referenced `WadRayMath::rayPow` implementation is not present in the Aave V3 codebase and is instead merged from the `WadRayMath` implementation of Aave V1. The Aave V1 implementation is compiled with a `pragma` statement of `^0.5.0`, rendering the statements within `rayPow` performed using unchecked arithmetics in the original implementation.

We advise the `rayPow` function's body to be wrapped in an `unchecked` code block, replicating the original behavior of `WadRayMath` in Aave V1. We would like to note that while presently not a vulnerability, code copied from other projects should execute in the same format it originates from.

Alex the Entreprenerd (judge) commented:

> **KBC-01L: Improper Disable of Initializer (Affected Lines: L30)**

> I would partially disagree on this, will award a Non-Critical.

> **KIB-01L: Improper Disable of Initializer (Affected Lines: L49)**

> See KBC-01L

> **KIB-02L: Insufficient Initial Epoch Sanitization (Affected Lines: L68)**

> Low because inconsistent.

> **KIB-03L: Discrepant Epoch Inclusivity Definitions (Affected Lines: L79-L81, L342-L345)**

> Low because inconsistent.

> **KIB-04L: Inexistent Enforcement of Minimums / Maximums in Yield (Affected Lines: L331)**

> Low

> **KAP-01L: Improper Disable of Initializer (Affected Lines: L39)**

> See KBC-01L

> **KFC-01L: Improper Disable of Initializer (Affected Lines: L32)**

> See KBC-01L

> **KFC-02L: Improper Release Event (Affected Lines: L88, L160)**

> Refactoring

> **KFC-03L: Inexistent Duplicate Entry Prevention (Affected Lines: L175-L180)**

> Duplicate of 13

> **KSP-01L: Inexistent Limitation of Variable Fee (Affected Lines: L360)**

> Low.

> **KSP-02L: Inexistent Limitation of Bond Fee (Affected Lines: L636-L643)**

> See KSP-01L

> **KSP-03L: Improper Disable of Initializer (Affected Lines: L78)**

> See KBC-01L.

> **KSP-04L: Unsafe Casting of Term Months (Affected Lines: L100)**

> Low.

> **BLT-01L: Inexistent Sanitization of State Transitions (Affected Lines: L38, L47)**

> Refactoring

> **KYC-01L: Weak Definition of Owner (Affected Lines: L49)**

> Low because trusted owner, but worth flagging.

> **KBT-01L: Potential Approval Blacklist Bypass (Affected Lines: L148)**

> Duplicate of 22.

> MAR-01L: Inexistent Sanitization of Maximum Answer (Affected Lines: L69-L72)

> Low.

> WRM-01L: Incorrect Code Merge (Affected Lines: L127-L137)

> Low, good flag IMO even in lack of exploit.

Alex the Entreprenerd (judge) commented:

> Best report by far, great work!

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/8

> **Status:** Mitigations confirmed with comments. Full details in reports from 0xsomeone, 0x52, and hihen.

Gas Optimizations

For this contest, 2 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **0xsomeone** received the top score from the judge.

The following warden also submitted a report: hihen.

`KBCToken.sol`

KBC-01G: Variable Data Location Optimization (Affected Lines: L51)

The `cBond` argument of the `KBCToken::issueBond` function is set as `memory` whilst the function itself is `external`. We advise it to be set to `calldata` optimizing the `issueBond` function's execution cost in `KUMASwap::buyBond`.

KBC-02G: Inefficient Variable Declaration (Affected Lines: L72)

The `KBCToken::redeem` function loads the full `CloneBond` struct into memory only to utilize the `cBond.parentId` member of it. We advise the `_bonds[tokenId].parentId` entry to be stored to a `uint256` variable that is consequently utilized, optimizing the function's gas cost significantly.

`KIBToken.sol`

KIB-01G: Unused Private Contract Member (Affected Lines: L40)

The referenced `_allowances` mapping present in the `KIBToken` implementation remains unused in the codebase. We advise it to be safely omitted as its presence can cause ambiguity with the homonym `_allowances` mapping of the `ERC20Upgradeable` implementation of OpenZeppelin.

KIB-02G: Redundant External Self Calls (Affected Lines: L142, L170, L276, L281)

The referenced statements perform an external self-call as they specify `this.balanceOf` instead of `balanceOf` directly, instructing the compiler to treat the statement as an external call to the `address(this)` target. We advise the code to utilize `balanceOf` directly, avoiding the significant gas overhead of external calls. If the code wishes to be verbose about which implementation it invokes, the `this.balanceOf` statements can be replaced by `KIBToken.balanceOf`, instructing the compiler to invoke the `KIBToken` implementation of `balanceOf` explicitly.

KIB-03G: Potential Arithmetic Optimizations (Affected Lines: L175, L280, L367)

The referenced arithmetic calculations are guaranteed to be performed safely by the conditional clauses whose execution precedes them. We advise them to be wrapped in `unchecked` code blocks, optimizing their execution. We should note that more calculations can theoretically be performed in `unchecked` code blocks (i.e. `block.timestamp - _lastRefresh` in `KIBToken::_calculateCumulativeYield`), however, the referenced lines are **guaranteed by conditionals rather than logic** to be performed safely and thus their validity is infallible.

`KUMAAddressProvider.sol`

KAP-01G: Redundant `Initializable` Import & Inheritence (Affected Lines: L14)

The `Initializable` contract is part of the `UUPSUpgradeable` inheritance chain and is also imported and inherited directly by `KUMAAddressProvider`. We advise it to be safely omitted from the contract's import list and direct inheritance declarations as it is ineffectual.

`KUMAFeeCollector.sol`

KFC-01G: Redundant `Initializable` Import & Inheritence (Affected Lines: L14)

The `Initializable` contract is part of the `UUPSUpgradeable` inheritance chain and is also imported and inherited directly by `KUMAFeeCollector`. We advise it to be safely omitted from the contract's import list and direct inheritance declarations as it is ineffectual.

KFC-02G: Potential Iterator Optimizations (Affected Lines: L165, L174, L208)

The referenced iterator increments / decrements are performed "safely" via Solidity's built-in safe arithmetic due to the usage of the `0.8.17` version compiler. We advise the relevant increment / decrement statements to be relocated to the end of their respective `for` loop and wrapped in an `unchecked` code block, optimizing their execution.

KFC-03G: Inefficient Loop Bounds (Affected Lines: L165-L166)

The referenced loop iterates from `payeesLength` to `1`, however, on each iteration it utilizes the `i` iterator by subtracting it by `1`. We advise the iteration to start from `payeesLength - 1`, apply no conditional for the loop, and contain an `if (i == 0) break;` statement at the end of the `for` loop, significantly optimizing the loop's gas cost.

KFC-04G: Inefficient Adjustment of Storage Variable (Affected Lines: L185)

The `_totalShares` variable is updated on each iteration of the `newPayees` array instead of being assigned to once at the end of the `KUMAFeeCollector::changePayees` function's execution. We advise a local `totalShares` / `totalShares_` variable to be declared that is incremented on each iteration of the `newPayees` array. Finally, we advise the `_totalShares` variable to be updated once after the `for` loop with the value of `totalShares` / `totalShares_`, requiring a single `SSTORE` operation for the function's execution.

KFC-05G: Potential Arithmetic Optimizations (Affected Lines: L136, L138)

The referenced arithmetic calculations are guaranteed to be performed safely by the conditional clauses whose execution precedes them. We advise them to be wrapped in `unchecked` code blocks, optimizing their execution. We should note that more calculations can theoretically be performed in `unchecked` code blocks (i.e. `_totalShares -= _shares[payee]` in `KUMAFeeCollector::removePayee`), however, the referenced lines are **guaranteed by conditionals rather than logic** to be performed safely and thus their validity is infallible.

KFC-06G: Inefficient Loop Limit Evaluation (Affected Lines: L208)

As loop limits in Solidity are dynamically evaluated when they contain function calls, the referenced `length` invocation will be repeated on each iteration of the `for` loop. We advise the `length` to be cached to a local variable that is consequently utilized for the `for` loop as it is not expected to change during the `KUMAFeeCollector::_release` function's execution.

KUMASwap.sol

KSP-01G: Unused Named Function Arguments (Affected Lines: L561)

The `KUMASwap::onERC721Received` function contains named function arguments yet does not utilize them. We advise their explicit names to be omitted. To note, a function signature is defined by the data types and not the variable names. As such, a `function` declaration of `onERC721Received(address, address, uint256, bytes calldata)` is entirely valid.

KSP-02G: Inefficient Loop Limit Evaluation (Affected Lines: L614)

As loop limits in Solidity are dynamically evaluated when they contain function calls, the referenced `length` invocation will be repeated on each iteration of the `for` loop. We advise the `length` to be cached to a local variable that is consequently utilized for the `for` loop as it is not expected to change during the `KUMASwap::_updateMinCoupon` function's execution.

KSP-03G: Potential Arithmetic Optimizations (Affected Lines: L587)

The referenced arithmetic calculations are guaranteed to be performed safely by the conditional clauses whose execution precedes them. We advise them to be wrapped in `unchecked` code blocks, optimizing their execution. We should note that more calculations can theoretically be performed in `unchecked` code blocks (i.e. `_couponInventory[bond.coupon]` in `KUMASwap::buyBond`), however, the referenced lines are **guaranteed by conditionals rather than logic** to be performed safely and thus their validity is infallible.

`KYCToken.sol`

KYC-01G: Non-Standard Override of Transfer Functionality (Affected Lines: L119-L121, L126-L132)

The `KYCToken` is meant to represent a non-transferrable token and this is achieved by overriding the relevant `transferFrom` and `safeTransferFrom` functions of the `ERC721` dependency of OpenZeppelin. However, the current mechanism only overrides `transferFrom(address, address, uint256)` and `safeTransferFrom(address, address, uint256, bytes memory)` and not `safeTransferFrom(address, address, uint256)`.

The `safeTransferFrom(address, address, uint256)` function invokes `safeTransferFrom(address, address, uint256, bytes memory)` in the code of OpenZeppelin and is protected, however, the current way of preventing transfers is non-standard and non-uniform. We advise the `_beforeTokenTransfer` hook to be overridden instead, ensuring that regardless of the implementations of `ERC721` the code will never permit a transfer as the `_beforeTokenTransfer` hook is guaranteed to be executed in all transfer flows including mint and burn operations.

`KUMABondToken.sol`

KBT-01G: Non-Standard Override of Transfer Functionality (Affected Lines: L185-L197, L205-L217)

The `KUMABondToken` is meant to represent a blacklist-able and pausable token and this is achieved by overriding the relevant `transferFrom` and `safeTransferFrom` functions of the `ERC721` dependency of OpenZeppelin. However, the current mechanism only overrides `transferFrom(address, address, uint256)` and `safeTransferFrom(address, address, uint256, bytes memory)` and not `safeTransferFrom(address, address, uint256)`.

The `safeTransferFrom(address, address, uint256)` function invokes `safeTransferFrom(address, address, uint256, bytes memory)` in the code of OpenZeppelin and is protected, however, the current way of preventing transfers is non-standard and non-uniform. We advise the `_beforeTokenTransfer` hook to be overridden instead, ensuring that regardless of the implementations of `ERC721` the code will apply proper access control to transfers as the `_beforeTokenTransfer` hook is guaranteed to be executed in all transfer flows including mint and burn operations.

KBT-02G: Inconsistent Usage of Caller / `_msgSender` (Affected Lines: L148, L156, L173, L176, L189, L193, L209, L213)

The codebase of `KUMABondToken` applies modifier-based access control on transfers and approvals by using the `msg.sender` value whilst the code of the implementations make use of the `_msgSender` GSN-compliant method. We advise the style to be standardized to either `msg.sender` or `_msgSender` across all instances, the former of which will lead to a gas optimization as it will not incur the gas overhead of invoking an internal function. To note, this does not constitute a vulnerability as the `_msgSender` implementation of the OpenZeppelin dependency in use yields the `msg.sender` value.

`MCAGRateFeed.sol`

MRF-01G: Redundant `Initializable` Import & Inheritence (Affected Lines: L13)

The `Initializable` contract is part of the `UUPSUpgradeable` inheritance chain and is also imported and inherited directly by `MCAGRateFeed`. We advise it to be safely omitted from the contract's import list and direct inheritance declarations as it is ineffectual.

MRF-02G: Conditional Optimization (Affected Lines: L92)

The referenced conditional within `MCAGRateFeed::getRate` will evaluate whether a `rate` is below the `_MIN_RATE_COUPON` threshold and yield the value of `_MIN_RATE_COUPON` instead in such a case. We advise the conditional to be made inclusive, optimizing the code's gas cost in the case of `rate == _MIN_RATE_COUPON`. This optimization is possible as inclusive and non-inclusive comparators utilize the same gas thus reducing the code's gas cost by one `JUMP` instruction if `rate == _MIN_RATE_COUPON`.

MRF-03G: Potential Arithmetic Optimizations (Affected Lines: L87, L89)

The referenced arithmetic calculations are guaranteed to be performed safely by the conditional clauses whose execution precedes them. We advise them to be wrapped in `unchecked` code blocks, optimizing their execution. The referenced lines are **guaranteed by conditionals rather than logic** to be performed safely and thus their validity is infallible.

`MCAGAggregator.sol`

MAR-01G: Improperly Empty Chainlink Format Variable (Affected Lines: L86)

The `MCAGAggregator::latestRoundData` function is meant to conform to the Chainlink paradigm and while it does fill in both the `roundId` and `answeredInRound` variables (the latter being irrelevant to the Kuma Protocol), the function does not fill in the `startedAt` value. We advise the `startedAt` value to be set to the same value as `updatedAt`, replicating the Chainlink paradigm and ensuring users of the `latestRoundData` can apply the same sanitizations regardless of whether they integrate with a Chainlink oracle or with the `MCAGAggregator`.

Alex the Entreprenerd (judge) commented:

> **KBC-01G: Variable Data Location Optimization (Affected Lines: L51)**

> 150

> **KBC-02G: Inefficient Variable Declaration (Affected Lines: L72)**

> Saves 3 slots from being loaded, that's 2.1k * 3
> 6.3k

> **KIB-01G: Unused Private Contract Member (Affected Lines: L40)**

> Valid QA but no meaningful savings

> **KIB-02G: Redundant External Self Calls (Affected Lines: L142, L170, L276, L281)**

> 100 per instance = 400

> **MAR-01G: Improperly Empty Chainlink Format Variable (Affected Lines: L86)**

> Not a savings finding, but definitely a good suggestion

> Would guesstimate the rest of the optimizations at around 50 / 100 gas each.

> Above 7k saved.

> Great report that is customized to the contest!

Alex the Entreprenerd (judge) commented:

> Example of what we want to see more of, amazing work!

m19 (KUMA) mitigated:

> https://github.com/code-423n4/2023-02-kuma/pull/9

> **Status:** Mitigations confirmed with comments. Full details in reports from 0xsomeone, 0x52, and hihen.

Mitigation Review

Introduction

Following the C4 audit contest, 3 wardens (0xsomeone, 0x52, and hihen) reviewed the mitigations for all identified issues. Additional details can be found within the C4 KUMA Versus Mitigation Review contest repository.

Overview of Changes

Summary from the Sponsor:

> This mitigation adds validation to state modifications and oracles in the KIBT, KUMASwap, MCAGRateFeed, and KUMABondToken contracts, in addition to gas optimizations and QA fixes.

Mitigation Review Scope

URL	Mitigation of	Purpose
https://github.com/code-423n4/2023-02-kuma/pull/3	H-01	This mitigation adds a check disallowing transfer to self in KIBT _transfer
https://github.com/code-423n4/2023-02-kuma/pull/4	M-01	This mitigation adds a check that the owner of a KUMABondToken approve call is not black listed
https://github.com/code-423n4/2023-02-kuma/pull/5	M-02	This mitigation adds a duplicate payees check in KUMAFeeCollector changePayees
https://github.com/code-423n4/2023-02-kuma/pull/6	M-03	This mitigation adds a staleness threshold check to MCAGRateFeed
https://github.com/code-423n4/2023-02-kuma/pull/7	M-04	This mitigation fixes the logic in KUMASwap sellBond revert for when maxCoupons has been reached
https://github.com/code-423n4/2023-02-kuma/pull/8	QA	This mitigation adds QA fixes - see PR for specific fixes

URL	Mitigation of	Purpose
https://github.com/code-423n4/2023-02-kuma/pull/9	GAS	This mitigation adds GAS fixes - see PR for specific fixes
https://github.com/code-423n4/2023-02-kuma/pull/10	term-fix	This mitigation refactors bond.term to months instead of seconds

Mitigation Review Summary

Of the mitigations reviewed, 7 have been confirmed:

- H-01: Mitigation confirmed with comments by 0xsomeone, 0x52, and hihen.
- M-01: Mitigation confirmed by 0xsomeone, 0x52, and hihen.
- M-02: Mitigation confirmed by 0xsomeone, 0x52, and hihen.
- M-03: Mitigation confirmed by 0xsomeone, 0x52, and hihen.
- M-04: Mitigation confirmed by 0xsomeone, 0x52, and hihen.
- QA: Mitigation confirmed with comments by 0xsomeone, 0x52, and hihen.
- Gas: Mitigation confirmed with comments by 0xsomeone, 0x52, and hihen.

The one remaining mitigation introduced a new issue. See full details below.

Non-fixed terms will cause variable length bonds with the same nominal yield to have different coupons

Submitted by 0x52

https://github.com/code-423n4/2023-02-kuma/blob/22fd56b3f0df71714cb71f1ce2585f1c4dd21d64/src/kuma-protocol/KUMASwap.sol#L177-L220

Vulnerability details

Note - The term refactoring has been made for the following reason:

```
Our main KIBT is intended to be backed by 1-year treasury bill
tokens, however, a bond issued on 1 Jan 2023 does not have the same
amount of seconds compared to a 1-year treasury bill issued on 1 March
2023, or 1 Jan 2024 etc.
```

This seems to be a misunderstanding because a 1-year treasury bill is not 1 year in length. It's actually a 52 week fixed length bond (source), so there's no need for a change. Second as explained below using the new system will cause compatibility issues for variable length bonds like treasury notes and other non-US bonds.

Impact

Non-fixed terms will cause variable length bonds with the same nominal yield to have different coupons

Proof of Concept

Currently coupon is used to track which bonds can and cannot be sold to KUMASwap. Coupon is the amount of yield the bond accumulates per second. Since term is now tracked in months rather than seconds the coupon rate will have to be different between nearly every bond issued. It will also hurt the compatibility of bonds because even when the underlying bond rate doesn't change, the coupon will have to be adjusted.

Example:

There are 31536000 seconds in a calendar year and there are 31622400 seconds in a leap year. This means that bonds that include the end of a leap year will have to have a different (and lower) coupon than bonds with the same yield that don't include the end of a leap year. This is because bonds that do will have a longer term and even though their yields are the same nominally (i.e. 3.5%) they're coupons will be different.

3.5% (non-leap year) => coupon = 1.00000000109085
3.5% (leap year) => coupon = 1.00000000108785

The result of this is that bonds that share the same nominal yields will be forced to have different coupons or else their value will be calculated incorrectly. This causes the frustrating problem that two bonds with the same nominal yield will have different coupons if one includes the end of a leap year while the other doesn't. Since bonds that have a coupon lower than the current rate can't be sold to KUMA swap these bonds can't be sold. This becomes increasingly complicated (and more fragmented) the longer the term of the bonds because different issuance dates will includes more or less leap year ends.

It is also worth considering that different types of bonds will behave differently. As an example T-Bills are fixed length bonds. The "1 year" T-Bill is actually a 52 week fixed length bond. The "2 year" T-Notes are variable length bonds which depends on leap years.

Recommended Mitigation Steps

Even though some bonds should have slightly longer terms (because of leap years) the previous way a tracking using fixed term length in seconds will provide a better user experience and prevent bonds with the same nominal yield from having different coupons.

m19 (KUMA) commented:

> This sparked some internal discussions and we are inclined to agree with the warden here. We've actually gone as far as reverting back the change completely.

> This helped us realize that our current model only works with fixed-length treasury bills, ie. 13 weeks, 26 weeks, 52 weeks because those are always the same duration. But as pointed out by the warden treasury notes such as the 2year notes are not fixed in length, we found several that were 730 or 731 or even 729 days in length. We will come up with a better solution in a v1.1 or v2 of the protocol to support these.

Alex the Entreprenerd (judge) commented:

> Am considering the coupon value as well as the duration inaccuracies for:

- Risk of value leak / mispricing
- Inconsistency in behavior between different bonds

m19 (KUMA) commented:

> @Alex the Entreprenerd - just a quick question: when you're talking about the risk of value leak/mispricing, you're talking about the `term-fix` branch changes or the original version?

> With the initial codebase (so ignoring the changes made in `term-fix`) we don't believe there is a leak of value risk, a different term in seconds would just result in different risk category contracts, and limiting it to fixed term treasury bills (many countries issue these, not just US, but also France for example) will solve this.

Alex the Entreprenerd (judge) commented:

> @m19 - Thank you for the info, in order to determine Severity, I'd have to estimate the risk introduced by the change (on the branch `term-fix`), we can then have a discussion around reverting the change as a recommendation.

> Would you say that `term-fix` introduced a leak of value or would you say it's mostly an inconvenience / inconsistency in behaviour?

m19 (KUMA) commented:

> @Alex the Entreprenerd - I do agree there could've potentially been a leak of value, for example, if we had started issuing 2-year bonds with a term of `24` while the actual bonds could've been 729, 730, or 731 days in duration. And that was our intention as per the PR description after all. This risk would've not been there if we had stuck with the term in seconds since the 729, 730 and 731 bonds would've all been in a different risk category.

> The changes to the `KUMASwap.sol` made it so that `bond.term` no longer has to be in seconds as it no longer was used for calculations and now could be anything (days, months, years) and thus allow us to group bonds of different durations together.

Alex the Entreprenerd (judge) commented:

> Per the discussion above, the finding has shown the following risk:

- Bonds with the same duration but different coupons
- Which could have caused KumaSwap to behave in an unintended way

> The sponsor is choosing to mitigate by changing `bond.term` to be expressed in an arbitrary unit which will help standardize and group bonds together.

> The finding has shown a new risk and inconsistent behavior that could have been introduced via the new PR, and for this reason I believe it meets the requirements for Medium Severity.

Disclosures

Top