SPEARBIT

# Tradable onchain v2 Security Review

**Auditors**

**Christoph Michel**, Lead Security Researcher

**0xIcingdeath**, Lead Security Researcher

**Cergyk**, Security Researcher

**Akshay Srivastav**, Associate Security Researcher

**Report prepared by:** Lucas Goiriz

September 2, 2024

# Contents

## 1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

## 2 Introduction

Tradable enables leading asset managers to safely and compliantly tokenize their strategies and access a rapidly growing and underserved global market for institutional grade yield products.

## 3 Risk classification

| **Severity level** | **Impact: High** | **Impact: Medium** | **Impact: Low** |
|---|---|---|---|
| **Likelihood: high** | Critical | High | Medium |
| **Likelihood: medium** | High | Medium | Low |
| **Likelihood: low** | Medium | Low | Low |

### 3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

### 3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

### 3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

## 4 Executive Summary

Over the course of 15 days in total, Tradable engaged with Spearbit to review the tradable-onchainv2 protocol. In this period of time a total of **36** issues were found.

**Summary**

| | |
|---|---|
| **Project Name** | Tradable |
| **Repository** | tradable-onchainv2 |
| **Commit** | aed8713a |
| **Type of Project** | DeFi |
| **Audit Timeline** | Jul 22nd to Aug 6th |

**Issues Found**

| **Severity** | **Count** | **Fixed** | **Acknowledged** |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 4 | 4 | 0 |
| Medium Risk | 5 | 4 | 1 |
| Low Risk | 6 | 5 | 1 |
| Gas Optimizations | 1 | 1 | 0 |
| Informational | 20 | 15 | 5 |
| **Total** | **36** | **29** | **7** |

# 5 Findings

## 5.1 High Risk

### 5.1.1 Bypass of `EntityType` check when null attribute is returned by Quadrata

**Severity:** High Risk

**Context:** QuadrataKYCVerifier.sol#L138-L148

**Description:** In `QuadrataKYCVerifier` when a deal's `EntityType` requirement is set to `Individual` or `Business` and Quadrata returns a null business attribute (all fields `bytes32(0)`) then the `EntityType` requirement check is bypassed.

The `EntityType` requirement check looks like this:

```
if (requirements.entityType != IQuadrataKYCVerifier.EntityType.Any) {
    bool isBusiness =
        businessAttribute.value.isBusinessEqual(true) && !_isExpired(investorAttribute,
↪   maxAllowedAttributeAge);
    if (requirements.entityType == EntityType.Individual && isBusiness) {
        revert EntityTypeCheckFailed(deal, account);
    }

    if (requirements.entityType == EntityType.Business && !isBusiness) {
        revert EntityTypeCheckFailed(deal, account);
    }
}
```

The issue arises when:

- An account's `didAttribute.value` is not `bytes32(0)`.
- `businessAttribute.value` is `bytes32(0)`.
- Then `businessAttribute.value.isBusinessEqual(true)` will equate to `false`.
- Due to the `&&` operation `_isExpired` function won't be called-
- Hence the `isBusiness` flag will become `false`.
- Due to which the next `if` blocks (L141 - L147) won't be executed.

Hence the `EntityType` requirement check gets bypassed for the account.

**Recommendation:** Consider validating that the `businessAttribute.value` is not `bytes32(0)`. Ideally this null value validation check should be performed for all queried attributes.

**Proof of concept:** This test case was added in `test/integration/quadrata-kyc-verifier/verifyEligibility.t.sol`:

```
function test_audit_emptyBusinessField() public {
    address _user = makeAddr("_user");

    quadrata.setDID(_user, attestationEpoch);
    quadrata.setCountry(_user, "US", attestationEpoch);
    quadrata.setAMLRiskScore(_user, 1, attestationEpoch);
    quadrata.setInvestorStatus(_user, true, attestationEpoch);
    // quadrata.setBusinessStatus(_user, false, attestationEpoch);

    quadrataKYCVerifier.verifyEligibility(dealToken, _user);
}
```

**Tradable:** Fixed in PR 23.

**Spearbit:** Fixed. Verification now reverts for null attributes.

### 5.1.2 AML requirement checks skipped if no AML attribute for user

**Severity:** High Risk

**Context:** QuadrataKYCVerifier.sol#L199

**Description:** The `_isExpired` function returns `false` when `attribute.epoch == 0` which happens when the attribute is missing and was not set by any Quadrata issuer.

```
/// Checks if an attribute is expired.
function _isExpired(
    IQuadPassportStore.Attribute memory attribute,
    uint256 maxAttributeAge
)
    internal
    view
    returns (bool)
{
    return attribute.epoch > 0 && attribute.epoch < block.timestamp - maxAttributeAge;
}
```

The AML check is implemented as

```
// Check: AML risk score.
if (
    !amlAttribute.value.amlLessThanEqual(requirements.maxAMLRiskScore)
        || _isExpired(amlAttribute, maxAllowedAttributeAge)
) {
    revert AMLCheckFailed(deal, account);
}
```

For a non-existent AML attribute, all `amlAttribute` values will be zero and the revert is never reached.

If a deal requires an AML score of 5 or below and the investor has not performed KYC/AML checks they will still be able to invest in the deal.

**Recommendation:** If any Quadrata attribute is missing but a requirement exists for the Deal, the `verifyEligibility` function should revert. Consider treating a null attribute (`attribute.epoch == 0`) as expire:

```
  /// Checks if an attribute is expired.
- function _isExpired(
+ function _isExpiredOrNull(
      IQuadPassportStore.Attribute memory attribute,
      uint256 maxAttributeAge
  )
      internal
      view
      returns (bool)
  {
-     return attribute.epoch > 0 && attribute.epoch < block.timestamp - maxAttributeAge;
+     return attribute.epoch == 0 || attribute.epoch < block.timestamp - maxAttributeAge;
  }
```

**Tradable:** Fixed in PR 23.

**Spearbit:** For Individual, this field will be omitted. If omitted means its value is `bytes32(0)` we will revert here and individuals cannot pass this check. Quadrata does omit the value for individuals (it will be `bytes32(0)`), it's unclear if `keccak256("FALSE")` will ever be set. The fix has been updated to account for that scenario.

### 5.1.3 `InvestmentManager::_deleteOffer` should subtract escrowAmount instead of `$.offers[id].amount`

**Severity:** High Risk

**Context:** InvestmentManager.sol#L360

**Description:** When an investor submits an offer to DealManager, the payment is initially escrowed, waiting for the offer to be reviewed by an Originator. If, after a delay of `offerEscrowPeriod` (initially `5 days`), the offer is not reviewed, the investor can call `withdrawOffer` to get his funds back.

However the accounting is incorrect in that case, since `amount` is deducted from the escrowed funds, whereas `escrowAmount` (amount + fees) has been escrowed.

The surplus amount (equivalent to the fee), stays locked in the contract.

> `hasSufficientAvailableFunds` would deduct `escrowedAmount` from current contract balance to determine the funds which can be distributed or withdrawn by admin.

**Recommendation:** Subtract `$.offers[id].escrowAmount` in InvestmentManager.sol:

```diff
  function _deleteOffer(uint256 id) private {
      IMStorage storage $ = _imStorage();
-     $.escrowedAmount -= $.offers[id].amount;
+     $.escrowedAmount -= $.offers[id].escrowAmount;
      assert($.offerIds.remove(id));
      delete $.offers[id];
  }
```

**Tradable:** Fixed in PR 14.

**Spearbit:** Fixed.

### 5.1.4 Deal tokens are mistakenly burnt for all fiat accounts on principal payout

**Severity:** Critical Risk

**Context:** PayoutManager.sol#L263-L267

**Description:** The principal payout distribution function can burn all fiat tokens unintentionally if the `payoutAmount` is set to a non-zero value, in a period where there are only fiat holders.

The function calculation uses the `_principalPayoutDistribution` function, which differentiates the fiat payment amount and on chain payment amount. Due to incorrect logic, the function incorrectly burns the deal tokens for all fiat accounts, where none should be burnt. If you take a scenario in which every current holder or yield recipient is a fiat account, the calculations in the relevant context mentioned above is:

```
fiatAccountsTotalBalance = fiat total balance
onchainAccountsTotalBalance = totalSupply - fiat total balance = 0
fiatPayoutAmount = onchainAccountsTotalBalance==0, therefore = 0
```

These are used to construct a list of payouts -- addresses and amounts of tokens that should be burnt. Notably, the `accountTokensToBurn` is going to continue to increase and the `principalPayouts` continue to grow in size.

```
function _principalPayoutDistribution(
    IDeal deal,
    uint256 payoutAmount
)
    internal
    view
    returns (Payout[] memory principalPayouts, uint256 netPayoutAmount, uint256 tokensToBurn)
{
    uint256 totalSupply = deal.totalSupply();
    if (totalSupply == 0) return (new Payout[](0), 0, 0);
    // Compute the fiat accounts total balance, to infer total payout amount.
    uint256 fiatAccountsTotalBalance = deal.fiatAccountsTotalBalance();
    uint256 onchainAccountsTotalBalance = totalSupply - fiatAccountsTotalBalance;
    uint256 fiatPayoutAmount = (onchainAccountsTotalBalance > 0)
        ? payoutAmount.mulDiv(fiatAccountsTotalBalance, onchainAccountsTotalBalance)
        : 0;

    // Calculate the total amount of tokens to burn.
    uint256 totalTokensToBurn = currencyToDealTokens(payoutAmount + fiatPayoutAmount);

    IDeal.TokenHolder[] memory dealHolders = deal.holders();
    uint256 length = dealHolders.length;

    principalPayouts = new Payout[](length);
    for (uint256 i = 0; i < length; i++) {
        IDeal.TokenHolder memory holder = dealHolders[i];
        // forgefmt: disable-next-item
        uint256 amount = holder.isFiatAccount ? 0 : holder.balance.mulDiv(payoutAmount,
↪   onchainAccountsTotalBalance);
        netPayoutAmount += amount;

        uint256 accountTokensToBurn = holder.balance.mulDiv(totalTokensToBurn, totalSupply);
        tokensToBurn += accountTokensToBurn;

        principalPayouts[i] = Payout({
            account: holder.account,
            amount: amount,
            burnTokenAmount: accountTokensToBurn,
            isFiatAccount: holder.isFiatAccount
        });
    }
}
```

The function is called in the `_initiatePrincipalPayout` which will then iterate over all the payouts and start burning the respective token amount for all payouts -- which in this case, would include the payouts for all fiat accounts.

**Scenario:** A deal is created, with only fiat holders. These newly created fiat accounts receives deal tokens in exchange for their fiat deposit. When the principal payout process starts, this function will burn all the on-chain deal tokens, despite the loan terms not ending. As a result, the principal and interest denominations between fiat accounts and on-chain accounts will also be mismatched.

**Recommendation:** The entire base of the system relies heavily on off-chain interactions with the DealAdmin -- and the mitigations as outlined below rely on the admin calls to be done in a timely manner according to the state of transfers off-chain. We recommend:

- Convert the `payoutAmount` to `totalPayoutAmount`, where the total payout amount represents the total amount to be burnt on-chain and off-chain.
- Burn `currencyToDeal(totalPayoutAmount)`
- Derive the on-chain payout and off-chain payout amounts from the total, since the contract already knows the distribution of funds between on-chain :: off-chain.

The above will allow the contracts to handle the zero-fiat balance case, as if there is no off-chain funds, then the assumption is all the funds will be on-chain funds; and vice versa. This allows an interest period to be closed, even if there are only off-chain fiat accounts.

**Tradable:** Fixed in PR 26 by recommendations outlined above.

**Spearbit:** Fixed.

## 5.2 Medium Risk

### 5.2.1 `DealManager::initiatePrincipalPayout` unbounded slippage on burnt tokens

**Severity:** Medium Risk

**Context:** PayoutManager.sol#L271

**Description:** A total payout amount is computed off-chain and an amount proportional to the on-chain only payment is passed as an argument to the call on `DealManager::initiatePrincipalPayout`. During execution, the total payout amount is reconstructed from the values of `onChainTotalBalance`, `fiatTotalBalance` and on chain payout. This total amount is used to determine the quantity of deal tokens to burn, but if the ratio has changed since the `payoutAmount` has been precomputed (some tokens are burnt or transferred), more tokens than needed will be burnt.

**Scenario:**

- **Initial state:**

```
onChainTotalBalances = 1000
fiatTotalBalances = 1000
```

  1. The originator computes a payout amount for a total payout of `1000`, since `onChainTotalBalances/totalSupply == 1/2`, payoutAmount passed as an argument to `initiatePrincipalPayout` is `500`.
  2. The state changes, and 500 tokens are burnt from a on chain account. As a result the state is:

- **Intermediate state_**

```
onChainTotalBalances = 500
fiatTotalBalances = 1000
```

  – The originator emits the computed transaction and as the ratios have changed, the total reconstructed payout is `1500` instead of `1000`, and instead of half the tokens, all of the tokens are burnt.

    Note that the scenario likelihood is yet to determine, since only an originator can do all of the operations outlined above (`initiatePrincipalPayout`, `burn` or `forceTransfer`).

**Recommendation:** Please consider introducing a slippage parameter for the burnt amount of tokens, and revert if more tokens than needed are about to be burnt.

**Tradable:** Fixed in PR 26.

**Spearbit:** Fixed.

### 5.2.2 Investors cannot receive their deposited principal or due interest if their deal requirement checks start failing

**Severity:** Medium Risk

**Context:** PayoutManager.sol#L234-L237, PayoutManager.sol#L68

**Description:** In `PayoutManager` contract the investor's eligibility check is performed when principal or interest is claimed for an investor.

- The `claimPayout` reverts if `checkAccountEligibility(msg.sender)` reverts.
- In `_pushPayout` the investor is skipped when `isEligibleAccount(account)` returns `false`.

There could be scenarios where an investor was eligible when he deposited principal in a deal but going forward some of his quadrata attributes become invalid as per the deal's requirement.

A simple example could be when a deal was set up to allow investors whose AML score is upto `3`. An investor with AML score of `3` invests into the deal. After a while the investor's AML score increases to `4`. Now neither that investor nor the protocol admin can pull out the investor's deposited principal and due interest. This leads to loss of funds for the investor.

**Recommendation:** Consider removing the eligibility checks for claiming payouts. Or consider adding a mechanism to recover funds of investors whose eligibility checks fail after investment.

**Tradable:** Fixed in PR 36. We decided to introduce a new function, that allows admin to revoke payout from the account that failed the eligibility check. In this case the funds will be returned to the deal originator for manual processing.

**Spearbit:** Fixed. In PR 36 a restricted `revokePayout` function has been implemented which sends the revoked funds to `capitalRecipient`.

### 5.2.3 `payoutPeriodStartTime` skips 1 second for payouts

**Severity:** Medium Risk

**Context:** PayoutManager.sol#L120

**Description:** When yield is being distributed via `PayoutManager._initiateInterestPayout`, the deal admin specifies a period end time. The period start time is taken as the last end time + 1 in `payoutPeriodStartTime`.

```
function payoutPeriodStartTime() public view returns (uint48) {
    PMStorage storage $ = _pmStorage();
    // If there are no interest payouts yet, return the yield generation start.
    if ($.latestInterestPeriodEnd > 0) {
        return $.latestInterestPeriodEnd + 1;
    }
    // ...
}
```

The idea is that the timeline is perfectly subdivided into consecutive periods so no period can be missed. However, the new periods start at `latestInterestPeriodEnd + 1`, skipping the 1-second period from `[latestInterestPeriodEnd, latestInterestPeriodEnd + 1]` each time when a yield distribution is made.

This leads to unfair distributions and in the worst case, a yield recipient will not receive their yield.

**Example 1:** Assume holders are entitled to yield from time `t + 0` to `t + 200` which hasn't been distributed yet. At `t+101` a new investor joins the deal and is minted deal tokens, entitling them to yield for the period `[t + 101, t + 200]`.

- First, yield is distributed up to `t + 100`.
- Afterwards, yield is distributed up to `t + 200`, note that this period is starting at `payoutPeriodStartTime = t+101` instead of `t+100`.
- The new investor will receive the same yield for the second distribution for the `[t+100, t+200]` period as an existing investor with the same balance. The existing investor should have received more yield for the second distribution as they held their deal token balance for 1 second longer (the period of `[t + 100, t + 101]`).

**Example 2:** The contracts are intended to be deployed on zkSync. The block time on zkSync is 1 second:

> An L2 block is generated every 1 second, encompassing all transactions received within that timeframe. zkSync docs

An investor joins the deal and is minted deal tokens at time `t + 0`. They exit the deal one second later at time `t + 1` (force transfer, principal payout, etc.). If the previous yield distribution ended at `t + 0` they won't receive any yield.

**Recommendation:** Consider changing the `payoutPeriodStartTime` to start exactly when the old period ended.

```
  function payoutPeriodStartTime() public view returns (uint48) {
      PMStorage storage $ = _pmStorage();
      // If there are no interest payouts yet, return the yield generation start.
      if ($.latestInterestPeriodEnd > 0) {
-         return $.latestInterestPeriodEnd + 1;
+         return $.latestInterestPeriodEnd;
      }

      return deal().yieldGenerationStart();
  }
```

**Tradable:** Fixed in PR 19.

**Spearbit:** Fixed.

### 5.2.4 `QuadrataKYCVerifier::verifyEligibility` wrong expiration check for `businessAttribute`

**Severity:** Medium Risk

**Context:** QuadrataKYCVerifier.sol#L140

**Description:** The following parameters are checked during KYC verification in QuadrataKYCVerifier.sol#L80-85:

```
// Query the account's attributes from Quadrata.
bytes32[] memory attributesToQuery = new bytes32[](5);
attributesToQuery[0] = QuadrataAttributes.DID;
attributesToQuery[1] = QuadrataAttributes.COUNTRY;
attributesToQuery[2] = QuadrataAttributes.AML;
attributesToQuery[3] = QuadrataAttributes.IS_BUSINESS;
attributesToQuery[4] = QuadrataAttributes.INVESTOR_STATUS;
```

And for each of these parameters, a deadline is checked to ensure that the attribute is not expired. Here is an example to check expiry for the business attribute:

```
bool isBusiness =
    businessAttribute.value.isBusinessEqual(true) &&
    !_isExpired(investorAttribute, maxAllowedAttributeAge); //@audit: check expiry, but checked on
↪   wrong attribute
```

Unfortunately the expiry is checked on `investorAttribute` instead of `businessAttribute` which would lead to wrong validation of an expired `businessAttribute`.

**Recommendation:** Fix the expiration attribute verification:

```
 function verifyEligibility(address deal, address account) external {
     QuadrataKYCVerifierStorage storage $ = _storage();

       // ... Unrelated KYC checks

     if (requirements.entityType != IQuadrataKYCVerifier.EntityType.Any) {
         bool isBusiness =
-            businessAttribute.value.isBusinessEqual(true) && !_isExpired(investorAttribute,
↪   maxAllowedAttributeAge);
+            businessAttribute.value.isBusinessEqual(true) && !_isExpired(businessAttribute,
↪   maxAllowedAttributeAge);
         console.logBool(isBusiness);
         if (requirements.entityType == EntityType.Individual && isBusiness) {
             revert EntityTypeCheckFailed(deal, account);
         }

         if (requirements.entityType == EntityType.Business && !isBusiness) {
             revert EntityTypeCheckFailed(deal, account);
         }
     }
 }
```

**Tradable:** Fixed in PR 15.

**Spearbit:** Fixed.

### 5.2.5 Contracts do not support tokens with fees or rebasing tokens

**Severity:** High Risk

**Context:** PayoutManager.sol#L73

**Relevant Context:** `DealManager.sol`, `InvestmentManger.sol`, `PayoutManager.sol`

**Description:** All these contracts assumes the payment currency is a standard-abiding ERC20, where there are no fees on transfer. Given Tradable Finance is considering DAI, USDC, and maybe USDT -- transfer amounts may not be exactly equal to expected values when transferring a token with fee.

For example, while an investor is answering a capital call, the actual amount sent to the capital recipient may not match the `cc.amount`, when a portion of the transfer is given to the owner of USDT. Moreover, the fee recipient will also receive a smaller amount than intended after feeAmount has been transferred. In this particular example, the deal tokens minted to the investor are based on the amount transferred, without considering the amount lost by fees, which can result in a mismatch between deal tokens and the amount a user should have received, with the fee taken into consideration.

```
/// @notice Answers a capital call.
/// @dev Handles the capital call answer and emits the `CapitalCallAnswered` event.
/// @param id The capital call ID.
function _answerCapitalCall(uint256 id) internal {
    [...]
    paymentCurrency.safeTransferFrom(msg.sender, capitalRecipient(), cc.amount);

    // Transfer fee amount from investor to fee recipient.
    if (cc.feeAmount > 0) paymentCurrency.safeTransferFrom(msg.sender, feeRecipient(), cc.feeAmount);

    // Finally mint the deal tokens to the investor.
    IDeal.Mint[] memory mints = new IDeal.Mint[](1);
    mints[0] = IDeal.Mint(msg.sender, currencyToDealTokens(cc.amount));
    deal().mint(mints);

    emit CapitalCallAnswered(id, cc.investor, cc.amount, cc.feeAmount);
}
```

This may also result in unexpected behaviour with rebasing tokens, where balances and totalSupply adjust, however does not seem to be as major of a concern, as Tradable is not considering adding these tokens.

**Exploit Scenario:** USDT with fee enabled is added into the system. When an investor answers a capital call and submits the funds for it, the capital recipient receives less than they asked for; and the investor receives more deal tokens than the amount that they actually submitted.

**Recommendation:** Do not add USDT into the system as a supported token. If there is a need to add USDT into the system, consideration must be added to account for the fees and the actual amounts that users receive and put into the system.

For other tokens, such as those that rebase or balances change, can have an egregious impact on the system, and should be avoided.

**Tradable:** Acknowledged.

**Spearbit:** Acknowledged.

## 5.3 Low Risk

### 5.3.1 `PayoutManager::_initiatePrincipalPayout` `totalTokensToBurn` can be greater than totalSupply due to roundings

**Severity:** Low Risk

**Context:** *(No context files were provided by the reviewer)*

**Description:** Due to multiple down roundings, the onchain payout amount can be slightly bigger than the maximum (e.g `totalTokensToBurn > totalSupply`).

**Scenario:**

-**Initial state:**

- Alice balance: 250 (onchain).
- Bob balance: 250 (onchain).
- Charlie balance: 250 (fiat).
- David balance: 250 (fiat).

Deal admin wants to initiate a full principal payout, so for a total of `1000`. The max value he should be able to provide as on chain `payout` should be `500`. Let's see what happens if the deal admin provides `501` instead:

1. `fiatPayoutAmount` is computed to be `501`:

```
uint256 fiatPayoutAmount = (onchainAccountsTotalBalance > 0) ?
    payoutAmount.mulDiv(fiatAccountsTotalBalance, onchainAccountsTotalBalance) :
    0
```

2. As a result `totalTokensToBurn` is computed to be `1002`.
3. Now iterating over accounts, and computing individual amounts to burn which are rounded down:

   Alice, Bob, Charlie, David: (250*1002)/1000 == 250

Since the individual amounts to burn are equal to the balances of each participant, the call succeeds.

**Recommendation:** Please consider adding an assert that `totalTokensToBurn <= totalSupply` in `PayoutManager::_principalPayoutDistribution`:

```
  // Calculate the total amount of tokens to burn.
  uint256 totalTokensToBurn = currencyToDealTokens(payoutAmount + fiatPayoutAmount);
+ assert(totalTokensToBurn <= totalSupply);
```

**Tradable:** Fixed in PR 26.

**Spearbit:** Fixed.

### 5.3.2 Open access of `verifyEligibility`, `checkAccountEligibility` & `isEligibleAccount` functions

**Severity:** Low Risk

**Context:** BaseManager.sol#L79, BaseManager.sol#L84, QuadrataKYCVerifier.sol#L65

**Description:** The protocol has these three open access functions:

- `QuadrataKYCVerifier.verifyEligibility`
- `DealManager.checkAccountEligibility`
- `DealManager.isEligibleAccount`

The functions perform an account's eligibility check based upon a deal's requirement.

The Quadrata's `QuadReader.getAttributesBulk` function has these statements:

```
bool hasPreapproval = governance.preapproval(msg.sender);
require(hasPreapproval, "SENDER_NOT_AUTHORIZED");
```

which shows that Quadrata exposes attributes to whitelisted readers only. Quadrata may whitelist Tradable's `QuadrataKYCVerifier` contract to read attribute data but Tradable is currently exposing that data to be read by all on-chain smart contracts.

Due to this, any on-chain contract can perform Quadrata based KYC check for an account on behalf of Tradable. Quadrata will always assume that the KYC check call is coming from Tradable protocol. In case Quadrata implements an access fee then Tradable can be forced to pay more fees than expected by abusing the mentioned open access functions.

**Recommendation:** The `DealManager.checkAccountEligibility` & `DealManager.isEligibleAccount` functions can be made private. The `QuadrataKYCVerifier.verifyEligibility` function can validate that the caller is a legitimate `DealManager` contract (registered in registry). `DealRegistry` can be updated to store `DealManager` addresses as well.

**Tradable:** Fixed in PR 39. The `DealManager.isEligibleAccount` public function is removed. `DealManager.checkAccountEligibility` is now protected and accessible to the read admin role only.

I've added caller validation to `QuadrataKYCVerifier.verifyEligibility`:

```
if (msg.sender != address(IDeal(deal).manager())) revert Unauthorized(msg.sender, deal);
```

This check happens after resolving the requirements. The assumption here is that having requirements set by the deal admin for the given deal, verifies that it is our deal and we can trust the `deal.manager()` return value.

**Spearbit:** Fixed. `QuadrataKYCVerifier.verifyEligibility` validates that the caller is a registered `DealManager`\n- `DealManager.checkAccountEligibility` validates that the caller is `DEAL_ADMIN` - `DealManager.isEligibleAccount` has been removed

### 5.3.3 Investors have no control over the amount of service fee they pay on interest

**Severity:** Low Risk

**Context:** PayoutManager.sol#L127-L129, PayoutManager.sol#L135-L140, PayoutManager.sol#L312

**Description:** The Tradable protocol has two fees

- Origination fee: it is the fee taken on amounts that investors invest.
- Service fee: it is the fee taken on amount of interest generated by the investments.

When investing into a deal the investors has the ability to set/view the max origination fee. But since the Tradable `DEAL_ADMIN` can change the service fee percentage using `_setServiceFee` during a deal's lifespan, the investors will never to sure about the max service fee they will pay at the end of deal term. Investors can be forced to pay higher service fee than they originally expected.

**Recommendation:** In `_setServiceFee` function consider enforcing that the new service fee cannot be greater than a max threshold value.

**Tradable:** Fixed in PR 28.

**Spearbit:** Fixed.

### 5.3.4 `BaseManager::__BaseManager_init` is missing `onlyInitializing` modifier

**Severity:** Low Risk

**Context:** BaseManager.sol#L46-L54

**Description:** The `__BaseManager_init` is the initialization function of abstract `BaseManager` contract which gets invoked in `DealManager.initialize` function.

Initialization functions must be only allowed to be invoked during the contract initialization. However the `BaseManager.__BaseManager_init` can currently be invoked at any point in time by a contract which inherits `BaseManager`.

**Recommendation:** Consider inheriting `Initializable` contract in `BaseManager` and add `onlyInitializing` modifier to `__BaseManager_init` function.

**Tradable:** Fixed in PR 25.

**Spearbit:** Fixed.

### 5.3.5 `yieldDistribution` iterates over unbounded `yieldRecipients` array

**Severity:** Low Risk

**Context:** Deal.sol#L359

**Description:** The `Deal.yieldDistribution` function is used whenever yield is distributed in `DealManager.initiateInterestPayout`. The function iterates over the entire `Deal`'s `$.yieldRecipients` state array.

This array never shrinks, yield recipients are only ever pushed onto it as soon as an account receives a balance. Note that while there is a `maxHoldersLimit` limit for the *current holders* (`$.currentHolders`), this limit does not apply to `$.yieldRecipients` as old holders are replaced by new holders (through `mint` and `burn` or `forceTransfer`).

Once the array becomes too large, the yield distribution's gas usage might not fit into a single block anymore.

**Recommendation:** As deal token holders are fully managed by the deal admin, there is some control over how large this array can grow.

> **Note:** technically the yield distributions do not need to happen for all yield recipients at once, it can be broken up into several calls. Long-term, `initiateInterestPayout` could specify an array of yield recipients to process if this becomes a problem. The transferred service fee must then be taken individually per yield recipient, in total relative to the paid out `netPayoutAmount` instead of the entire `payoutAmount` each time.

**Tradable:** Acknowledged.

**Spearbit:** Acknowledged by the client.

### 5.3.6 `InvestmentManager::_issueCapitalCall` Capital calls are silently overridden

**Severity:** Low Risk

**Context:** InvestmentManager.sol#L188

**Description:** Capital calls can be used by an originator to ask for investors to provide more funds. However when one capital call has already been issued for an investor, issuing another one to the same investor will silently override the existing capital call.

There are two mappings to keep track of issued capital calls:

- `capitalCalls`
- `investorCalls`

When a new capital call is added over an existing one, the value in `investorCalls` is updated for the investor, whereas a new value is simply added for `capitalCalls` at the new id.

> This means that in the case 2 capital calls have been added for an investor, the investor can answer any of those 2 capital calls, but only the most recent one will be shown by the view functions `capitalCalls(address investor)`, `capitalCalls()`

**Recommendation:** Delete previously issued capital call when issuing a new one:

In `InvestmentManager.sol`:

```
  function _issueCapitalCall(
      address investor,
      uint256 amount,
      uint48 dueDate
  )
      internal
      returns (CapitalCall memory cc)
  {
      if (investor == address(0)) revert CapitalCallInvestorZeroAddress();
      if (dueDate < Time.timestamp()) revert CapitalCallPastDueDate();

      // Validate the investment amount & calculate the origination fee.
      _checkInvestmentAmount(amount);
      uint256 feeAmount = calculateOriginationFee(amount);

      IMStorage storage $ = _imStorage();

+     //Delete previous capital call, if any exists
+     (bool exists, uint256 prevId) = $.investorCalls.tryGet(investor);
+     if (exists) {
+         _deleteCapitalCall(prevId);
+     }
+
      uint256 id = $.nextCapitalCallId++;
      cc = CapitalCall({ id: id, investor: investor, amount: amount, feeAmount: feeAmount, dueDate:
↪   dueDate });
      $.capitalCalls[id] = cc;
      $.investorCalls.set(investor, id);

      emit CapitalCallIssued(id, investor, amount, feeAmount, dueDate);
  }
```

**Tradable:** Fixed in PR 16.

**Spearbit:** Fixed.

## 5.4 Gas Optimization

### 5.4.1 Unnecessary second binary search for `_yieldAt`

**Severity:** Gas Optimization

**Context:** Deal.sol#L534

**Description:** The `Deal._yieldAt` function looks at the closest checkpoints left and right of the search timestamp. It first performs a binary search to find the closest checkpoint older or at the search timestamp (`upperCheckpointLookup`). Afterwards, a second binary search is performed to find the closest checkpoint newer than the search timestamp (`lowerCheckpointLookup`).

**Recommendation:** The second checkpoint (if it exists) is always the next checkpoint after the older one (after `upperCheckpointLookup`), i.e., if `pos` is the index of the first checkpoint, `pos+1` will be the index of the second checkpoint that is currently retrieved by `lowerCheckpointLookup`.

A function could be added to `CheckpointsLib` that also returns the index for `upperCheckpointLookup`. Then add a `nextCheckpoint(index)` function that returns the `(bool nextExists, uint48 nextTimestamp, uint208 nextValue)` tuple.

**Tradable:** Fixed in PR 21. Synced the Checkpoints library with the OZ upstream, implemented the suggested changes and also removed the `lowerCheckpointLookup`.

**Spearbit:** Fixed.

## 5.5 Informational

### 5.5.1 Assumptions made by off-chain smart contracts

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The smart contract system relies heavily on the DealAdmin calling functions frequent enough to keep the smart contracts as a single source of truth. The below highlight assumptions that we have made of the behaviour in the off-chain system during this review:

- The `DealAdmin` chooses when to mint deal tokens and must be matched timely to off-chain actions:
  - Tradable has specific requirements of interest accrual to specific parties while off-chain wire transfers are in flight, thus the `DealAdmin` must only mint deal tokens when a party receives their respective tokens, and when they want interest for different parties to start accruing.
  - Tradable must also correctly mint and burn tokens, especially for fiat-accounts, where the number of tokens they hold is representative of an off-chain value.
- Tradable determines when deals are deployed, and must make sure that any off-chain contracts are signed before on-chain interaction begins.
- Tradable determines when to issue a capital call to an investor on behalf of a user requesting said capital call, and is trued to put the correct amount and due date on-chain.
- Tradable is trusted such that it should not cancel a capital call that was not intended to be cancelled.
- Tradable updates whatever necessary on-chain data is required in order to have a well-informed decision about the state of a deal. This can include the price, the net asset value, and other fields.

**Recommendation:** Expand this list to enumerate through all the inputs that the smart contracts rely on from the off-chain components, and ensure that all off-chain components cover these respective risks to mitigate a misconfigured deal or mismatch in state from off-chain data to smart contract data.

**Tradable:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.5.2 `PayoutManager::_initiatePrincipalPayout` Edge case preventing all supply to be burnt

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The function `initiatePrincipalPayout` can be used to repay investors of a deal, and burn all of the deal tokens, if the repayment is full. However there is an edge case which would prevent burning all of the tokens when some tokens have been burnt already, and which is due to difference of precision between underlying and deal token.

**Scenario:**

- paymentCurrency is USDC with 6 decimals.
- Alice has an `on chain` account.

1. Admin issues a capital call for Alice. Alice mints for a deposit of 1000 USDC, 1000 deal tokens (`1000e18`).
2. Admin burns `(5e18 - DUST)` deal tokens from Alice. Alice balance in deal tokens is now `(995e18 + DUST)`.
3. Admin is unable to issue a full principal payout for Alice, because of the difference in precision between USDC (6 decimals) and deal token (18 decimals). This is due to the fact that the number of tokens to burn from Alice is determined from the payout amount denominated in USDC, and which is then scaled up using `currencyToDealTokens`:

```
// Calculate the total amount of tokens to burn.
uint256 totalTokensToBurn = currencyToDealTokens(payoutAmount + fiatPayoutAmount);
```

So by providing the max amount allowed: `995 USDC`, Alice keeps `DUST` of deal tokens.

**Recommendation:** Impact is minimal, so it may be acceptable to leave as is. Alternatively, some remediations can be:

- Only enable mints and burns in multiples of `10**(18-paymentCurrencyDecimals)`.
- Specify exact amount to burn, and check inside `initiatePrincipalPayout` that.

**Tradable:** Fixed in PR 26.

**Spearbit:** Fixed. If `dust = scalingFactor` (`1e12` in case of USDC) would remain after principal payout burn, this is now burned as well.

### 5.5.3 `DEAL_MAX_INVESTMENT_AMOUNT` check is superfluous

**Severity:** Informational

**Context:** InvestmentManager.sol#L344

**Description:** `DEAL_MAX_INVESTMENT_AMOUNT` is set to be `type(uint128).max` which is not a realistic value for limiting investment size.

There are two distinct cases where this check is used:

- Issuing a capital call, in which case the `DEAL_ADMIN` can provide an arbitrary amount, because no funds are transferred at that time. In this case the check can prevent some amount values, but there are still unreasonable amounts (below `type(uint128).max`) which would be accepted.
- Submitting an offer, in this case the amount is bounded by the balance of the submitter, since funds are transferred straight away.

**Recommendation:** If this check is to be used defensively against a reasonable scenario, the value needs to be set lower, and/or be configurable (currently it is a constant).

**Tradable:** Fixed in PR 34. We decided to drop this cap as redundant. There is no risk of overflow, since all the investments amounts are in stable coins, which are USD equivalents.

**Spearbit:** Fixed.

### 5.5.4 `Deal::price` function can return incorrect or outdated price values

**Severity:** Informational

**Context:** Deal.sol#L209-L223, IDeal.sol#L20-L28

**Description:** The price value of deal tokens is determined by `net asset value / total supply`. Since NAV needs to be updated manually by admin. In case there is a lag in the NAV updates then `price` will return incorrect price.

For example, suppose that initially Nav = 100 & supply = 100 so price is 1. After some time 100 more deal tokens get minted so now price will be 100/200 = 0.5 (assuming a delay in NAV update). Incorrect price will be returned. Similarly opposite scenario occurs when deal tokens are burned and NAV update is delayed.

The function also returns the most recent timestamp between `supplyUpdateTimestamp` & `nav.timestamp`. When the above explained delay scenario happens, the function will also return the more recent `supplyUpdateTimestamp` timestamp further tricking the price consumer into trusting the returned price.

The current natspec of `Price.timestamp` type says it is `The price latest update timestamp`.

**Recommendation:** As NAV of a deal (principal + interest) changes as soon as any deal tokens get minted or burned. It must be made sure that on-chain NAV state is updated as soon as deal token supply changes.

Also it is not optimal to return the latest of `supplyUpdateTimestamp` & `nav.timestamp` as this can trick price consumers about the last update timestamp. The `price` function can simply return the `nav.timestamp`.

**Tradable:** The NAV and the price calculation will be addressed in the next iteration of the project. The price is not used onchain by any of the existing contracts. For now we just acknowledge the current state and recommendations.

**Spearbit:** Acknowledged.

### 5.5.5 `_beaconProxyBytecodeHash` depends on specific compiler used

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** zkSync uses a different bytecode hash format and the deployments work differently, the bytecode for a bytecode hash must be known beforehand.

The `_beaconProxyBytecodeHash` function returns the bytecode hash as the word at offset of 36 bytes of `type(BeaconProxy).creationCode`.

```
function _beaconProxyBytecodeHash() private pure returns (bytes32 bytecodeHash) {
    bytes memory creationCode = type(BeaconProxy).creationCode;
    assembly {
        // +32 to skip the length of creationCode, + 36
        bytecodeHash := mload(add(creationCode, 68))
    }
}
```

**Recommendation:** We could not find a specification or a guarantee that the bytecode hash can always be found in the creation code at this offset and it seems highly solc/zksolc compiler-version-specific. Ensure deployments are tested whenever a new compiler version is used.

**Tradable:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.5.6 `for` loop iteration for contracts can be unbounded

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The contracts contain functions that loop over lists that can grow significantly in size. Once these contracts store enough data, this can potentially run out of gas.

For example, the `DealRegistry` function has a list function that iterates through all the deals that have been created to return the address of all deals. Once the list of deals grows significantly, this function can become problematic:

- `DealRegistry.sol#L80-L90`:

```
/// @inheritdoc IDealRegistry
function list() external view returns (address[] memory) {
    EnumerableMap.Bytes32ToAddressMap storage deals = _storage().deals;
    uint256 length = deals.length();
    address[] memory result = new address[](length);
    for (uint256 i = 0; i < length; i++) {
        (, address deal) = deals.at(i);
        result[i] = deal;
    }
    return result;
}
```

The same list is iterated over in `DealPriceEngine::dealPrices()` to calculate the prices of all deals.

- DealPriceEngine.sol#L73-L79:

```
function dealPrices() external view returns (DealPrice[] memory) {
    address[] memory dealAddresses = _storage().dealRegistry.list();
    uint256 length = dealAddresses.length;
    DealPrice[] memory prices = new DealPrice[](length);
    for (uint256 i = 0; i < length; i++) {
        prices[i] = _price(IDeal(dealAddresses[i]));
    }
```

**Recommendation:** Create an off-chain monitoring system that captures when new deals are created, and saves their addresses in a database off-chain. This would allow this same off-chain service to query the price of every deal through the backend instead of being implemented on the smart contract.

**Tradable:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.5.7 Missing short-circuit to halt yield period calculation for periods with no checkpoints

**Severity:** Informational

**Context:** Deal.sol#L504-L507

**Description:** If the contract has not accrued any yield, the `_yieldForPeriod` will needlessly call the `_yieldAt` function when it could instead, short circuit and just return 0 directly.

The `_yieldForPeriod` function uses the `_yieldAt` function which immediately returns zero if there are no checkpoints. In essence, the `_yieldAt` function will be called twice, both times returning zero, and the `_yieldForPeriod` will just return a result of 0.

```
function _yieldAt(
    Checkpoints.Trace208 storage checkpoints,
    uint48 timestamp,
    uint256 tokenBalance
)
    private
    view
    returns (uint256)
{
    // If there are no checkpoints, return 0.
    if (checkpoints.length() == 0) return 0;
```

Given the fact that the `yieldDistribution` function iterates over the yield period for every single recipient in the for loop below, it would save on logic and gas costs to halt as early as a 0 period is known.

```
/// @notice Returns the yield distribution for a period.
/// @dev The yield is distributed pro rata based on the deal historical holding share of each holder in
↪  the period.
/// @dev CAUTION: This function is gas-intensive and should be used with caution in transactional
↪  contexts.
/// @param periodStartTime The start timestamp.
/// @param periodEndTime The end timestamp.
function yieldDistribution(
    uint48 periodStartTime,
    uint48 periodEndTime
)
    public
    view
    assertValidPeriod(periodStartTime, periodEndTime)
    returns (YieldRecipient[] memory yieldRecipients)
{
    DealStorage storage $ = _storage();
    yieldRecipients = new YieldRecipient[]($.yieldRecipients.length());
    uint256 recipientsCount = 0;
    for (uint256 i = 0; i < yieldRecipients.length; i++) {
        address account = $.yieldRecipients.at(i);
        uint256 yield =
            _yieldForPeriod($.accountYieldCheckpoints[account], periodStartTime, periodEndTime,
↪   balanceOf(account));

        if (yield > 0) {
            // forgefmt: disable-next-item
            yieldRecipients[recipientsCount++] = YieldRecipient({
                account: account,
                yield: yield,
                isFiatAccount: $.fiatAccounts.contains(account)
            });
        }
    }

    if (recipientsCount < yieldRecipients.length) {
        assembly {
            mstore(yieldRecipients, recipientsCount)
        }
    }
}
```

**Recommendation:** Add a check that if `endValue == 0`, return 0 for the `_yieldForPeriod` function instead of needing to recall `_yieldAt` for the start time.

**Tradable:** Fixed in PR 33.

**Canitna Managed:** Fixed.

### 5.5.8 Using assertions for state can mask root cause investigation for transactions

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The system uses `assert` statements to differentiate state that should not be possible, or is as a result of an invariant violation. The issue with this approach is the incredibly challenging and involved debugging process required to determine the cause of an assertion failure. This is why many fuzzing suites or codebases that rely heavily on assertion failures will implement helper functions to add messages to assertions to identify where certain lines have failed.

Examples of this can be seen throughout the codebase, including the validation for the number of tokens for the payment currency:

- `BaseManager.sol#L46-L72`:

```
function __BaseManager_init(
    IDeal deal_,
    IERC20Metadata paymentCurrency_,
    IKYCVerifier kycVerifier_,
    address capitalRecipient_,
    address feeRecipient_
)
    internal
{
    if (address(deal_) == address(0)) revert DealZeroAddress();
    if (address(paymentCurrency_) == address(0)) revert PaymentCurrencyZeroAddress();

    // Retrieve payment currency decimals to ensure correct scaling in calculations.
    uint8 paymentCurrencyDecimals = paymentCurrency_.decimals();

    // Ensure deal token and payment have compatible decimals.
    assert(paymentCurrencyDecimals <= 18 && deal_.decimals() == 18);

    BaseManagerStorage storage $ = _bmStorage();
    $.deal = deal_;
    $.paymentCurrency = paymentCurrency_;
    $.scalingFactor = 10 ** (18 - paymentCurrencyDecimals);

    _setKYCVerifier(kycVerifier_);
    _setCapitalRecipient(capitalRecipient_);
    _setFeeRecipient(feeRecipient_);
}
```

The withdrawal funds to capital recipient uses an assert to check that the capital recipient is non-zero:

- `DealManager.sol#L65-69`:

```
/// @inheritdoc IBaseManager
function withdrawFundsToCapitalRecipient(uint256 amount) external restricted
↪  hasSufficientAvailableFunds(amount) {
    address recipient = capitalRecipient();
    assert(recipient != address(0));
    paymentCurrency().safeTransfer(recipient, amount);
}
```

It is also used to check the "*success*" of the addition or removal of `offerIds` in `InvestmentManager::_submitOffer`, `InvestmentManager::_deleteCapitalCall` and `InvestmentManager::_deleteOffer`.

**Recommendations:** Convert the `DealManager.withdrawFundsToCapitalRecipient` and `BaseManager.__BaseManager_init` functions to use custom errors, as these are effectively data validation in the system.

**Tradable:** Fixed in PR 32 by following recommendations.

**Spearbit:** Fixed.

### 5.5.9 `PayoutManager::_initiateInterestPayout` should revert if `yieldGenerationStart` is zero

**Severity:** Informational

**Context:** PayoutManager.sol#L123

**Description:** `DealManager::initiateInterestPayout` can be called even if no deal tokens have ever been emitted for the deal, and this would generate a yield period going from the timestamp 0 to `periodEndTime`. Although it does hinder any functionality of the protocol, this can cause bugs in tools displaying yield period data, since there would be an overly large first period.

**Recommendation:** Please consider reverting in `PayoutManager::payoutPeriodStartTime`:

```
  /// @inheritdoc IPayoutManager
  function payoutPeriodStartTime() public view returns (uint48) {
      PMStorage storage $ = _pmStorage();
      // If there are no interest payouts yet, return the yield generation start.
      if ($.latestInterestPeriodEnd > 0) {
          return $.latestInterestPeriodEnd + 1;
      }

-     return deal().yieldGenerationStart();
+     uint yieldGenerationStart = deal().yieldGenerationStart();
+     if (yieldGenerationStart == 0) {
+         revert YieldGenerationNotStarted();
+     }
+     return yieldGenerationStart;
  }
```

**Tradable:** Fixed in PR 31.

**Spearbit:** Fixed.

### 5.5.10 Incorrect yield can be distributed due to insufficient period validity checks of `assertValidPeriod`

**Severity:** Informational

**Context:** Deal.sol#L79-L85

**Description:** The `Deal::assertValidPeriod` modifier looks like this:

```
modifier assertValidPeriod(uint48 periodStartTime, uint48 periodEndTime) {
    if (periodStartTime >= periodEndTime) revert DealYieldInvalidPeriod(periodStartTime, periodEndTime);
    if (periodEndTime < yieldGenerationStart()) revert DealYieldPastLookup(periodEndTime,
↪   yieldGenerationStart());
    if (periodStartTime >= clock()) revert DealYieldFutureLookup(periodStartTime, clock());
    _;
}
```

Also note that `yieldGenerationStart` function can return `0` in case no global checkpoint has been created yet.

Currently the `assertValidPeriod` modifier doesn't sufficiently validate the `periodStartTime` & `periodEndTime` timestamps which can lead to incorrect yield distributions. Some scenarios are listed below:

1. `periodStartTime` can be `0` when `yieldGenerationStart` returns `0` .
2. `periodStartTime` can be any value between `[0, block.timestamp - 1]`.
3. `periodEndTime` can be any value between `[yieldGenerationStart, type(uint48).max]`.

As the `Deal._yieldAt` function performs extrapolation of yield checkpoints, the scenario 3 becomes more severe.

Accounts can receive extrapolated yields for a future timestamp T, deal tokens of those accounts can be burned, transferred or increased before the actual timestamp T occurs, which leads to incorrect yield distribution.

**Recommendation:** Change the current implementation to this:

```
modifier assertValidPeriod(uint48 periodStartTime, uint48 periodEndTime) {
    if (periodStartTime >= periodEndTime) revert DealYieldInvalidPeriod(periodStartTime, periodEndTime);
    if (yieldGenerationStart() == 0) revert /*...*/;
    if (periodStartTime < yieldGenerationStart()) revert /*...*/;
    if (periodStartTime >= clock()) revert DealYieldFutureLookup(periodStartTime, clock());
    if (periodEndTime > clock()) revert /*...*/;
    _;
}
```

**Tradable:** Fixed in PR 31.

**Spearbit:** Fixed. In PR 31 necessary checks has been added to `PayoutManager._interestPayoutDistribution`.

### 5.5.11 Capital calls can be issued to non KYCed users

**Severity:** Informational

**Context:** InvestmentManager.sol#L169-L181

**Description:** Currently the `DEAL_ADMIN` can issue capital calls to non-KYCed (ineligible) users. Though those ineligible users cannot answer any capital call. This is because the `_issueCapitalCall` is missing `checkAccountEligibility` call.

**Recommendation:** Just as `amount` and `dueDate` are validated in `_issueCapitalCall`, the eligibility of `investor` parameter must also be validated.

**Tradable:** Fixed in PR 29.

**Spearbit:** Fixed.

### 5.5.12 `Deal::_yieldAt` - Missing safecasting the extrapolated yield checkpoint value

**Severity:** Informational

**Context:** Deal.sol#L536

**Description:** In `_yieldAt` when the next checkpoint does not exist then the next checkpoint value is extrapolated assuming linear growth.

```
if (!nextExists) return prevValue + ((timestamp - prevTimestamp) * tokenBalance);
```

As `tokenBalance` is an `uint256` the `(timestamp - prevTimestamp) * tokenBalance` is also casted to `uint256` by default.

Ideally, following the checkpoint convention throughout the `Deal` contract the `(timestamp - prevTimestamp) * tokenBalance` value should be safecasted to `uint208` before performing any other arithmetic operation.

**Recommendation:** Add these changes:

```
- if (!nextExists) return prevValue + ((timestamp - prevTimestamp) * tokenBalance);
+ if (!nextExists) return prevValue + SafeCast.toUint208((timestamp - prevTimestamp) * tokenBalance);
```

**Tradable:** Fixed in PR 21.

**Spearbit:** Fixed.

### 5.5.13 Missing input validations in `QuadrataKYCVerifier::setDealRequirements`

**Severity:** Informational

**Context:** QuadrataKYCVerifier.sol#L166-L169

**Description:** The `setDealRequirements` simply writes the input `DealKYCRequirements` parameter to storage without validating its individual parameters.

This creates two issues:

1. Empty countries array can be set as `DealKYCRequirements.allowedCountries`. In `verifyEligibility` execution is reverted if a deal's `DealKYCRequirements.allowedCountries` is an empty array.
2. `DealKYCRequirements.maxAMLRiskScore` of `0` can be set as a deal requirement. Since Quadrata returns a number `[1 - 10]` as AML risk score, a `0` value will make the deal unusable.

**Recommendation:** In `setDealRequirements` consider validating that:

- `requirements.allowedCountries.length` is not `0`.
- `requirements.maxAMLRiskScore` is not `0`.

**Tradable:** Fixed in PR 27.

**Canitna Managed:** Fixed.

### 5.5.14 Assert correct `Deal` and `DealManager` factory deployment

**Severity:** Informational

**Context:** DealFactory.sol#L133-L149

**Description:** The `DealFactory` precomputes the beacon proxy addresses for the `Deal` and `DealManager` as there's a cyclic dependency between their addresses. However, it does not check if the actual deployed addresses match the precomputed ones.

**Recommendation:** As the deployment is on zkSync and address derivation is done differently there and zkSync recommends to thoroughly test deployments as the bytecode needs to be known beforehand, consider asserting that the addresses match after deployment.

Also note that only the `dealManagerAddress` needs to be precomputed, the actual deployment address of the `Deal` contract, which is deployed first, can then be used when deploying the `DealManager`.

> Unfortunately, it's impossible to differentiate between the above cases during compile-time. As a result, we strongly recommend including tests for any factory that deploys child contracts using `type(T).creationCode`. zkSync Docs

**Tradable:** Fixed in PR 22.

**Spearbit:** Fixed.

### 5.5.15 Typos, Documentation & Unused Code

**Severity:** Informational

**Context:** Checkpoints.sol#L184, Constants.sol#L15, Deal.sol#L103, DealFactory.sol#L101

**Description:** There are typos and documentation errors in the code:

1. Checkpoints.sol#L184: "Return the index of the last (most recent) checkpoint with key lower or equal than the search key". This function returns the first index (oldest) that is strictly greater than the search key. The usage is correct and no code adjustments are required. (This has also been reported to the upstream OZ repo).

2. Deal.sol#L103: The `Deal` contract's authorization is done by checking if the caller is the `DealManager` contract. It does currently not require `AccessManagedUpgradeable` with the authority checks. Consider removing the dependency for `Deal`.
3. Constants.sol#L15 The `PUBLIC` role is currently not used in the contracts and deployment scripts.
4. DealFactory#L101 Natspec for `DealFactory::deployDeal` specifies: `The deal ID is used as the salt for the create2 address`. Which is currently not the case.
5. ACL.md: Rename `autorizeUpgrade` to `authorizeUpgrade`.

**Recommendation:** Consider addressing the mentioned issues.

**Tradable:**

- 1 is fixed in PR 21.
- 2 & 5 are fixed in PR 30.
- 3: I would like to keep `PUBLIC` role constant for visibility. It is a valid role which can be used with `AccessManager`.
- 4 is fixed in PR 22.

**Spearbit:** Fixed. About the 3) I think it's also a good idea to keep it for readability, so it is clear that public is not the default.

### 5.5.16 Add explicit `periodStartTime < periodEndTime` check to `_initiateInterestPayout`

**Severity:** Informational

**Context:** PayoutManager.sol#L190

**Description:** When `PayoutManager._initiateInterestPayout` distributes yield, it's important that no period can be distributed twice. As the code sets the next period start to the previous period's end time, it's important that no period end in the past can be specified. The `periodStartTime < periodEndTime` check is already performed downstream in `_interestPayoutDistribution` → `Deal.totalYield` → `assertValidPeriod`.

**Recommendation:** Consider explicitly performing this check in the `PayoutManager._initiateInterestPayout` function as well, to make it more obvious that periods cannot be reset.

**Tradable:** Fixed in PR 20.

**Spearbit:** Fixed.

### 5.5.17 Use internal setters to initialize contract state

**Severity:** Informational

**Context:** InvestmentManager.sol#L58, QuadrataKYCVerifier.sol#L59-L60

**Description:** Some contracts set contract state directly in their constructor or initialization function even though there is a setter function for the state.

1. InvestmentManager.sol#L58: Use `_setOfferEscrowPeriod`.
2. QuadrataKYCVerifier.sol#L59-L60: Use an internal `_setQuadrataReader` and `_setAttributeMaxAge` function that the external `setQuadrataReader` and `setAttributeMaxAge` will also use.

**Recommendation:** Consider using the setter function so the corresponding event is emitted even for the first contract state values.

**Tradable:** Fixed in PR 18.

**Spearbit:** Fixed.

### 5.5.18 Rename `__CapitalCallManager_init` to `__InvestmentManager_init`

**Severity:** Informational

**Context:** InvestmentManager.sol#L54

**Description:** The `InvestmentManager`'s initialization function is currently called `__CapitalCallManager_init`.

**Recommendation:** Consider renaming it to `__InvestmentManager_init` as the initialization function should match the contract name.

**Tradable:** Fixed in PR 17.

**Spearbit:** Fixed.

### 5.5.19 Using `currencyToDealTokens` and `dealTokensToCurrency` may result in rounding errors

**Severity:** Informational

**Context:** BaseManager.sol#L110-L118

**Description:** These two functions are expected to be inverses of each other. However, the rounding directions specified here can result in this inverse assumption failing. For example, take a simple example such as a currency amount of `1e18+1`:

`*currencyToDeal function -- * currencyAmount = 1e18 + 1`

```
= (1e18+1) * 1e18
= 1e36+1e18
```

`*dealTokensToCurrency function -- * dealTokenAmount = 1e36+1e18`

```
= (1e36+1e18)/1e18
= 1e36/1e18 + 1e18/1e18
= 1e18
```

For larger values, `x/scalingFactor` will be truncated when downscaling which can cause unexpected behaviour. However, since `dealTokensToCurrency` is not used in this codebase, this is marked as an "*informational*" issue.

**Recommendation:** Either remove the `dealTokenToCurrency` function, or this function is needed:

- When money is going into the protocol (usually to convert to deal tokens), round up.
- When money is exiting the protocol (to transfer to users), round down.

**Tradable:** Fixed in PR 13.

**Spearbit:** Fixed.

### 5.5.20 Fiat/onchain account state must be set prior to distribution

**Severity:** Informational

**Context:** Deal.sol#L163

**Description:**Adding/removing a fiat account can increase/decrease `fiatAccountsTotalYield` if the account had a balance during that period. This will in turn decrease/increase the `onchainAccountsTotalYield` computed in `PayoutManager._interestPayoutDistribution`. The yield payout can be wrong when it uses the wrong baseline which can happen in the following cases.

1. A fiat account Deal holder is not set as a fiat account in `Deal` when the yield distribution happens.
2. An onchain account Deal holder is set as a fiat account in `Deal` when the yield distribution happens.
3. An account received Deal tokens from both fiat and onchain payments. The account's balance is always fully counted as either fiat or onchain, never partially. The onchain payout parameter `payoutAmount` must also fully account the balance for either offchain or onchain payment.

**Recommendation:** When distributing yield for a period, all Deal token holders during that period need to have their proper account state (fiat / onchain) set prior to distribution, and the `DealManager.initiateInterestPayout`'s onchain payout parameter `amount` must have been computed based on these account states.

**Tradable:** Acknowledged. This is in fact the intended behavior. Ensuring the correct fiat account status prior the distribution and communicating this information to the originator is deal admin responsibility, and will be handled offchain.

**Spearbit:** Acknowledged.