Huma Finance Security Review

Auditors

0xLeastwood, Lead Security Researcher

Saw-Mon and Natalie, Lead Security Researcher

Kankodu, Security Researcher

Jonatas Martins, Associate Security Researcher

Report prepared by: Lucas Goiriz and Jonatas Martins

March 12, 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

Huma V2 is an on-chain lending protocol targeting business use cases. It is designed to run on EVM-compatible chains. The smart contracts are designed to be adaptive by supporting plugins for tranche policies, fee managers, and due managers

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of huma-contracts-v2 according to the specific commit. Any modifications to the code will require a new security review.

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 13 days in total, Huma Finance engaged with Spearbit to review the huma-contracts-v2 protocol. In this period of time a total of **81** issues were found.

Summary

Project Name	Huma Finance
Repository	huma-contracts-v2
Commit	70e779...7981bd
Type of Project	Lending Protocol
Audit Timeline	Jan 08 to Jan 24
Two week fix period	Jan 24 - Feb 07

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	2	2	0
Low Risk	28	25	3
Gas Optimizations	11	9	2
Informational	38	33	5
Total	**81**	**71**	**10**

5 Findings

5.1 High Risk

5.1.1 Restriction of `transfer` can be circumvented by using `approve` + `transferFrom`

Severity: High Risk

Context: TrancheVault.sol#L494, FirstLossCover.sol#L271

Description: In `TrancheVault` and `FirstLossCover` , `transfer` is disabled. However, those inherits from `ERC20Upgradeable` which includes `approve` and `transferFrom` functions that can circumvent this restriction.

Here are the following issues that may arise through the use of `approve` + `transferFrom`:

1. `TrancheVault` shares, intended only for addresses with the `LENDER_ROLE`, can be sent to an address without this role. A trustless smart contract can be built to make this functionality available as well.
2. The `checkLiquidityRequirementForRedemption` check can be bypassed by approving and calling `transferFrom()` to a new account not associated with `poolOwnerTreasury` or `evaluationAgent`, followed by redeeming these shares.
3. Lender's principal amount can be converted into yield which is readily redeemable, circumventing the redemption process. This is done by transferring the principal amount from a reinvesting lender's account to a non-reinvesting lender's account. The excess amount is treated as yield by the `processYieldForLenders` function.

Recommendation: Consider disabling the `transferFrom` function as well.

Huma: Fixed in PR 387 and PR 384.

Spearbit: Fixed.

5.1.2 `_getStartOfNextHalfYear` does not update `year` correctly for the second half of the year

Severity: High Risk

Context: Calendar.sol#L208

Description: `year` is not incremented for the second half of the year when we want to calculate the `startOfNextHalfYear`:

```
(uint256 year, uint256 month, ) = DTL.timestampToDate(timestamp);
startOfNextHalfYear = DTL.timestampFromDate(year, month > 6 ? 1 : 7, 1);
```

If `month > 6` is `true` one should also increment the `year`.

Recommendation: One should instead do:

```
function _getStartOfNextHalfYear(
    uint256 timestamp
) internal pure returns (uint256 startOfNextHalfYear) {
    (uint256 year, uint256 month, ) = DTL.timestampToDate(timestamp);
    if (month > 6) {
        month = 1;
        ++year;
    } else {
        month = 7;
    }
    startOfNextHalfYear = DTL.timestampFromDate(year, month, 1);
}
```

Also unit test cases missing checking this condition. At least 2 more test cases need to be added. Here is one by providing a `0` timestamp:

```
diff --git a/test/unit/common/CalendarTest.ts b/test/unit/common/CalendarTest.ts
index 4fbcd4d..1cd0bbb 100644
--- a/test/unit/common/CalendarTest.ts
+++ b/test/unit/common/CalendarTest.ts
@@ -782,6 +782,28 @@ describe("Calendar Test", function () {
                     ),
                 ).to.equal(startDateOfNextPeriod.unix());
             });
+
+            it("Should return the start date of the immediate next period relative to the current
↪  block timestamp if timestamp is 0 and the next period is in the next year", async function () {
+                const nextYear = moment.utc().year() + 1;
+                const nextBlockTime = moment.utc({
+                    year: nextYear,
+                    month: 10,
+                    day: 2,
+                });
+                await mineNextBlockWithTimestamp(nextBlockTime.unix());
+
+                const startDateOfNextPeriod = moment.utc({
+                    year: nextYear + 1,
+                    month: 0,
+                    day: 1,
+                });
+                expect(
+                    await calendarContract.getStartDateOfNextPeriod(
+                        PayPeriodDuration.SemiAnnually,
+                        0,
+                    ),
+                ).to.equal(startDateOfNextPeriod.unix());
+            });
         });
     });
```

run:

```
yarn hardhat test --network hardhat --grep "Calendar Test"
```

Huma: Good catch, this is a bug that we will fix, however, the severity should not be high risk since we do not expect this function to be called in real life, thus extremely low likelihood. Since we focus on short-duration loans, no period will be longer than one month. This semi-annual period feature is added purely for future expansion. Fixed in PR 403.

Spearbit: Fixed.

5.2 Medium Risk

5.2.1 First loss cover fee investment will fail if it does not exceed minimum deposit amount

Severity: Medium Risk

Context: PoolFeeManager.sol#L309-L311, FirstLossCover.sol#L363-L365

Description: Fees are invested in two different ways. Whenever an admin account attempts to withdraw accrued income and alternatively via `investFeesInFirstLossCover()` which is managed by the pool owner and sentinel service accounts. Until the cover reaches max liquidity, fees will be invested.

There is important edge case that is not considered which would cause `_investFeesInFirstLossCover()` to revert, affecting all fee withdrawal functions. When `getAvailableCap() < poolSettings.minDepositAmount`, then `depositCoverFor()` will fail and unless cover redeemability is enabled, it will not be possible to withdraw fees because `_investFeesInFirstLossCover()` will always attempt to deposit an amount that is less than `poolSettings.minDepositAmount`.

Recommendation: Consider making an exemption for `depositCoverFor()` that would allow for deposits to be made that are less than `poolSettings.minDepositAmount`.

Huma: Fixed in PR 460.

Spearbit: Verified fix.

5.2.2 Lenders can grief other lenders by depositing on their behalf

Severity: Medium Risk

Context: TrancheVault.sol#L274-L282, TrancheVault.sol#L538-L562, TrancheVault.sol#L300-L306

Description: Upon depositing into the tranche vault, the `deposit()` function will check that both the `msg.sender` and `receiver` accounts are approved to LP into the tranche. The deposit record keeps track of the principal amount held by the account, and the last deposit timestamp.

To redeem tranche shares, the lender must add a redemption request which is only processed at the beginning of each new epoch (assuming there is funds available to process all requested shares). In an effort to prevent depositors from timing their deposits whenever borrowers are due to make repayments, a withdrawal lockout period is enforced within `addRedemptionRequest()`.

Therefore, honest users can have their redemptions blocked by malicious actors if they deposit on their behalf before a new request is created. This resets the withdrawal lockout and can be repeated to prevent all tranche users from exiting.

```
function addRedemptionRequest(uint256 shares) external {
    // ...
    // Checks against withdrawal lockup period.
    DepositRecord memory depositRecord = _getDepositRecord(msg.sender);
    if (
        nextEpochStartTime <
        depositRecord.lastDepositTime +
            poolConfig.getLPConfig().withdrawalLockoutPeriodInDays *
            SECONDS_IN_A_DAY
    ) revert Errors.WithdrawTooEarly();
    // ...
```

Note: the cost for maintaining the attack is `poolSettings.minDepositAmount` every `poolConfig.getLPConfig().withdrawalLockoutPeriodInDays * SECONDS_IN_A_DAY`.

Recommendation: Create an internal approval system that allows for lenders to designate approved lenders. This ensures each lender has full control over who is allowed to deposit assets on their behalf.

Huma: We will get rid of the `receiver` parameter so that lenders can only deposit for themselves, not others. The "deposit for others" functionality has been removed in PR 433.

Spearbit: Fixed.

5.3 Low Risk

5.3.1 Inconsistent `FirstLossCover.isSufficient` check in `PoolFeeManager`

Severity: Low Risk

Context: PoolFeeManager.sol#L107-L125

Description: Admin accounts accrue fees in the `PoolFeeManager` contract whenever a profit distribution is made. These functions only allow each of the admin accounts to withdraw fees that are in excess of the first loss cover's max liquidity. This check is inconsistently applied across all withdraw functions and proves unnecessary as fees can only be invested via `investFeesInFirstLossCover()` up until min liquidity is reached. A pool owner or sentinel service account must intervene to ensure fees are invested.

Recommendation: Consider removing the sufficient cover checks in `withdrawPoolOwnerFee()` and `withdrawEAFee()` as `_investFeesInFirstLossCover()` will already attempt to invest fees up to the cover's max liquidity which can be assumed to be greater than min liquidity.

Huma: We have followed the original suggestion of adding the `isSufficient()` check to `withdrawProtocolFees()`. Addressed in PR 390.

Spearbit: Fixed.

5.3.2 First loss cover fee mechanics can be gamified

Severity: Low Risk

Context: FirstLossCover.sol#L167-L189

Description: The first loss cover contracts receive a portion of profits generated by the pool. FLC liquidity is bounded between min and max amounts and any excess yield is processed and paid out as yield to all cover providers via `payoutYield()`. Because cover can be readily deposited and redeemed, it seems trivial for cover providers to take on no risk by depositing up to `maxLiquidity` prior to any pool profit distributions and subsequently redeeming their tokens.

However, because liquidity is capped, other cover providers may be more willing to take on actual risk. If `maxLiquidity` has been reached, then this gamified deposit/redeem strategy will not longer be possible.

Recommendation: Similar to `TrancheVault`, it may be useful to add a time delay to any cover redemptions. Alternatively, `redeemCover()` could be disabled altogether unless the `pool.readyForFirstLossCoverWithdrawal()` flag has been set.

Another point to note, `poolSettings.minDepositAmount` can be circumvented because any cover amount can be redeemed after depositing.

Huma: We are disabling FLC withdrawal altogether unless the `readyForFirstLossCoverWithdrawal` flag is true, and under normal circumstances it's only set to true on pool closure. See PR 408.

Spearbit: Verified fix.

5.3.3 `depositCoverFor` does not check if `receiver` is a cover provider

Severity: Low Risk

Context: FirstLossCover.sol#L146-L156

Description: While the `depositCoverFor()` function is only called by `PoolFeeManager` contract when investing admin income into the FLC contracts, there is no check to ensure these admin accounts are even cover providers in the first place. In the case where they are not, they will not be eligible for any yield paid out.

Recommendation: It may be useful to check if the `receiver` account is already a cover provider and have them added if they are not. The `PoolFeeManager` contract is already trusted in terms of managing these accounts so it may make sense to extend this trust to managing the set of cover providers.

Huma: Fixed in PR 442.

Spearbit: Fixed.

5.3.4 `_processEpoch` should check against `minPoolBalanceForRedemption` before processing junior tranche token requests

Severity: Low Risk

Context: EpochManager.sol#L239-L245

Description: The `minPoolBalanceForRedemption` variable is initialized and set to avoid rounding errors when the pool has a low balance. Prior to redeeming senior tranche tokens, this is checked, however, this is not done before processing the junior tranche.

Recommendation: Add the same early return line prior to processing any junior tranche token requests.

```
if (availableAmount <= minPoolBalanceForRedemption) return;
```

Huma: Fixed in PR 382.

Spearbit: Fixed.

5.3.5 `_availableCredits` does not get updated when a receivable is burnt or transfered

Severity: Low Risk

Context: ReceivableBackedCreditLineManager.sol#L98-L106, ReceivableBackedCreditLineManager.sol#L67-L73, ReceivableBackedCreditLine.sol#L223, Credit.sol#L234, Credit.sol#L574

Description:

1. In `_approveReceivable` we have the following invariant enforced:

```
availableCredit <= cc.creditLimit // per borrower
```

$$0 \leq \sum_{approve} r^{i}_{adv} \cdot a_i - \sum_{drawdown} b_j \leq c_{lim}$$

2. In `_prepareForDrawdown` we have the following invariant enforced:

```
amount <= receivable.receivableAmount
```

$$b_j \leq a_j$$

3. In `_drawdown` the following invariant is enforced:

```
borrowAmount <= (cc.creditLimit - cr.unbilledPrincipal - (cr.nextDue - cr.yieldDue))
```

$$b_j \leq c_{lim} - (p_{un} + a_{next} - y_{due}) = c_{lim} - \sum_{i \neq j} b_i$$

Putting all these together we have:

$$-c_{lim} + b_j \leq \sum_{approve} r^i_{adv} \cdot a_i$$

- $\sum_{drawdown, i \neq k} b_k$ - $c_{lim} \leq b_j$

and

$$b_j \leq \min \left(a_j, c_{lim} - \sum_{i \neq j} b_i, \sum_{approve} r^i_{adv} \cdot a_i - \sum_{drawdown, i \neq k} b_k \right)$$

Now if you get a receivable approved and then burn it and you get another receivable approved the last element in the `min` function would have a higher value:

$$\sum_{approve} r^i_{adv} \cdot a_i - \sum_{drawdown, i \neq k} b_k$$

Since in the first sum $r^i_{adv} \cdot a_i$ term from the burnt receivable is still incorporated.

This is because when burning a `tokenId` for a receivable `_availableCredits` for a `borrower` does not get updated/decreased.

Recommendation: The fact that `_availableCredits` doest not get updated when a receivable is burnt or transferred should either be fixed or documented.

Huma: First of all, the creation of a receivable does not automatically lead to its use in approval and drawdown processes. In instances where a receivable remains unused for these purposes, the `_availableCredits` remains unaffected.

Furthermore, in the scenario where a receivable, let's say `A`, is approved but subsequently burned by the borrower, it's true that a technical adjustment to `_availableCredits` might be warranted. However, this situation is mitigated by the fact that the borrower must resort to a new receivable, referred to as `B`, for any future borrowing. Since the borrowing limit is constrained by the value of `B`, any credits previously approved under `A` become redundant and non-beneficial, thus disincentivizing the borrower from such actions.

Finally, implementing the update feature introduces non-trivial complexity. The current design features a unidirectional relationship where the `Credit` contract is aware of the `Receivable` contract. To accommodate the proposed feature, a bidirectional relationship between these two contracts would be necessary. Given the intricacies involved, the ROI for this change seems minimal.

Added comments in PR 458.

Spearbit: Fixed by documenting the code.

5.3.6 `onERC721Received` should revert when protocol is paused or the pool is off

Severity: Low Risk

Context: ReceivableBackedCreditLine.sol#L54-L61, ReceivableFactoringCredit.sol#L104-L111

Description: `onERC721Received` does allow receiving NFTs even when the protocol is paused or the pool is off. This is unlike other operational endpoints.

Recommendation: Make sure `onERC721Received` reverts when either the protocol is paused or the pool is off by adding:

```
poolConfig.onlyProtocolAndPoolOn();
```

Huma: Fixed in PR 463 by applying the suggestion.

Spearbit: Fixed.

5.3.7 `approveReceivable` does not check `receivableId` to make sure it is not zero

Severity: Low Risk

Context: ReceivableFactoringCreditManager.sol#L86

Description: The check to make sure `receivableInput.receivableId` is not `0` is missing.

Recommendation: Add a sanity check to make sure `receivableId` is not zero.

Huma: Fixed in PR 432.

Spearbit: Fixed.

5.3.8 `setPoolUnderlyingToken` does not validate the `_underlyingToken`

Severity: Low Risk

Context: PoolConfig.sol#L400, HumaConfig.sol#L40

Description: `setPoolUnderlyingToken` does not validate the `_underlyingToken`.

Recommendation: Make sure `setPoolUnderlyingToken` checks against the `validLiquidityAssets` before setting the `underlyingToken`.

Huma: We will get rid of the `setPoolUnderlyingToken()` function since swapping out the underlying token while the pool is active is operationally complex and error-prone. If we need to change the underlying token, we will close the pool and start a new one from scratch. To address the issue you raised, we will post a PR to remove this function.

Fixed in PR 434.

Spearbit: Fixed.

5.3.9 Only the huma master admin should be allowed to update `poolFeeManager` in `PoolConfig`

Severity: Low Risk

Context: PoolConfig.sol#L361-L366

Description: An `account` with an `DEFAULT_ADMIN_ROLE` can rewrite `poolFeeManager`. Pool fee manager is in charge of setting the protocol fee among other fees.

Recommendation: It might be best to only allow the huma master owner to change `poolFeeManager`.

Huma: Since the `PoolFeeManager` is more about the pool owner and EA fees than protocol fees, we think it's still the pool owner's responsibility to manage it. While it's true that they may swap in a version that does away with the protocol fee, the impact is small because:

1. The protocol fee will be small in magnitude, so the pool owner isn't really incentivized to do so just for the small gain and risk themselves being banned from ever using the protocol.
2. The Huma owner has the power to swap it back if needed.

Spearbit: Acknowledged.

5.3.10 Make sure only the huma master admin can call `setHumaConfig`

Severity: Low Risk

Context: PoolConfig.sol#L369-L374

Description: An `account` with a `DEFAULT_ADMIN_ROLE` can rewrite `humaConfig` in `PoolConfig`. But in terms of power structure, this should not be allowed, only the huma master admin should be able to perform this task.

Recommendation: Make sure only the huma master admin can call `setHumaConfig`.

Huma: Fixed in PR 431.

Spearbit: Fixed.

5.3.11 `seniorLoss` is not compared to `seniorTotalAssets` to make sure only the minimum value is subtracted

Severity: Low Risk

Context: Pool.sol#L351-L358

Description: `seniorLoss` is not compared to `seniorTotalAssets` to make sure only the minimum value is subtracted.

Recommendation: Like `juniorLoss` one calculate `seniorLoss` as below:

```
uint256 seniorTotalAssets = assets.seniorTotalAssets;
uint256 seniorLoss = seniorTotalAssets > loss - juniorLoss ? loss - juniorLoss : seniorTotalAssets;
```

Huma: Fixed in PR 395.

Spearbit: Fixed.

5.3.12 `setReinvestYield` should be restricted to a `lender` with `LENDER_ROLE`

Severity: Low Risk

Context: TrancheVault.sol#L201-L216

Description: Currently the `setReinvestYield` endpoint does not have any restrictions as to what `lenders` are allowed to be supplied by the pool operator. If an account does not have the `LENDER_ROLE` anymore, pool operator might be mistake try to add them to the `nonReinvestingLenders` list.

Recommendation: `setReinvestYield` should be restricted to a `lender` with `LENDER_ROLE`.

Huma: Fixed in PR 426.

Spearbit: Fixed.

5.3.13 Liquidity checks slightly differ around the edge cases in `FirstLossCover.redeemCover`

Severity: Low Risk

Context: FirstLossCover.sol#L177-L178, FirstLossCover.sol#L183-L184

Description: The two conditional statements for the `if` blocks in this context slightly vary in their edge cases:

```
if (!ready &&          currTotalAssets <= minLiquidity) { ... }
if (!ready && assets > currTotalAssets  - minLiquidity) { ... }
```

The second `if` block is equivalent to:

```
if (!ready && currTotalAssets - assets < minLiquidity) { ... }
```

The first `if` block would not allow the operation to be performed even when `currTotalAssets == minLiquidity`, but the second `if` block allows the updated total assets to be equal to `minLiquidity`.

Recommendation: It would be best to make sure the liquidity checks are consistent around the edge cases for these checks.

Huma: We are disabling withdrawal from FLC altogether if the `PoolReadyForFLCWithdrawal` flag is `false` in PR 408.

Spearbit: Fixed.

5.3.14 Non-reinvesting Lenders can front-run removal and still in `nonReinvestingLenders` array

Severity: Low Risk

Context: TrancheVault.sol#L102

Description: There's an issue where a Lender in `nonReinvestingLenders` can front-run the `removeApprovedLender()` function. By doing this, they can opt-out but still receive yield payouts (at the cost of reduced shares) until `setReinvestYield()` is called. Hence there is no relevant impact it is still a minor issue.

Recommendation: It's recommended to remove the check for `hasRole()`, the `_revokeRole()` function already checks for the role and and confirms if a role is revoked, in this case, it's not necessary to validate the return.

```
  function removeApprovedLender(address lender) external {
      poolConfig.onlyPoolOperator(msg.sender);
      if (lender == address(0)) revert Errors.ZeroAddressProvided();
-     if (!hasRole(LENDER_ROLE, lender)) revert Errors.LenderRequired();
      _revokeRole(LENDER_ROLE, lender);
      if (!_getDepositRecord(lender).reinvestYield) {
          _removeLenderFromNonReinvestingLenders(lender);
      }
  }
```

Huma: Fixed in commit b64298d5.

Spearbit: Fixed.

5.3.15 Be intentional about existence of `makeInitialDeposit` by restricting deposits from lenders if totalSupply is 0

Severity: Low Risk

Context: TrancheVault.sol#L123

Description: There is a `makeInitialDeposit` function that accepts deposits from authorized initial depositors. In a deposit that is meant for all the lenders, this is not enforced by making sure `totalSupply` is non-zero.

Recommendation: Add a check in the deposit function to ensure that `totalSupply` is greater than 0.

Huma: Fixed in PR 448.

Spearbit: Fixed.

5.3.16 Lenders can perform an inflation attack on the tranche vault

Severity: Low Risk

Context: TrancheVault.sol#L538-L562, TrancheVault.sol#L289-L352

Description: While a minimum deposit amount is enforced when an LP deposits assets into the tranche vault, the same user is free to redeem any amount of assets.

An inflation attack could then be setup in the following way:

- Deposit `poolConfig.getPoolSettings().minDepositAmount` into the protocol.
- Wait `poolConfig.getLPConfig().withdrawalLockoutPeriodInDays * SECONDS_IN_A_DAY` seconds.
- Create a redemption request to redeem all shares except for `1 wei`.
- Subsequently, wait for a new lender to deposit assets and perform the typical inflation-style attack by front-running the deposit to increase the `shares:assets` exchange rate, ensuring the victim receives zero shares at the end of their deposit.

Note: because the attacker must hope that no other approved lenders deposit assets into the tranche until they have redeemed all of their shares except for `1 wei`, this attack is unlikely to be seen.

Recommendation: Have the pool owner deposit a minimum amount before enabling a pool to lenders.

Huma: Fixed in PR 415.

Spearbit: Verified fix.

5.3.17 Uninitialized `FirstLossCover` in `PoolFactory`

Severity: Low Risk

Context: PoolFactory.sol#L303-L322

Description: The current implementation in `PoolFactory.sol` involves setting a new `FirstLossCover` without initializing it within the same transaction. This approach introduces a vulnerability where an uninitialized state can be exploited by frontrunners, potentially leading to a DoS attack on the deployed contract. Furthermore, this could disrupt the intended behavior of the `Pool` contract, resulting in operational issues or security risks.

Recommendation: It is recommended to initialize the `FirstLossCover` within the same transaction when it's set in `PoolFactory.sol`. This can be achieved by calling the initialization function of the `FirstLossCover` immediately after its creation by setting it in `setFirstLossCover()` function.

Huma: We added the initialize in PR 417, meanwhile rewrote the`setFirstLossCover` function to make internal and external calls consistent.

Spearbit: Fixed.

5.3.18 Redemption implementation does not match the protocol specification

Severity: Low Risk

Context: TrancheVault.sol#L635-L672

Description: The specification outlines that a LP's shares are minted in proportion to their total ownership of shares held by the pool and not the total requested share amount. However, the implementation is of the latter, meaning a lender who holds `X` shares and desires to redeem only 10% of this amount, can optimise this redemption by requesting to redeem a much more significant amount. This can be readily done prior to an epoch being closed as the available funds to be distributed is already known.

Following redemption execution, the same lender can cancel the redemption of any excess shares and will have gamified the redemption process in their favour.

Recommendation: Consider documenting this behaviour if it is accepted as a protocol weakness. Otherwise, make the necessary adjustments to distribute funds in proportion to the total percentage of the lender's requested shares.

Huma: Protocol spec updated and added the reasoning why we decided to implement it this way in the last paragraph under the linked section in the docs.

Spearbit: Fixed.

5.3.19 Blacklisted liquidity providers may cause `processYieldForLenders` to revert

Severity: Low Risk

Context: TrancheVault.sol#L421-L444, PoolSafe.sol#L44-L49

Description: Many stablecoins maintain blacklists that frequently block accounts that are suspected of illicit activity. If a non-reinvesting lender is blacklisted, then it is no longer possible to process yield for other non-reinvesting lenders until they are forced to become a reinvesting lender.

```
function processYieldForLenders() external {
    uint256 len = nonReinvestingLenders.length;

    uint256 price = convertToAssets(DEFAULT_DECIMALS_FACTOR);
    uint96[2] memory tranchesAssets = pool.currentTranchesAssets();
    for (uint256 i = 0; i < len; i++) {
        address lender = nonReinvestingLenders[i];
        uint256 shares = ERC20Upgradeable.balanceOf(lender);
        uint256 assets = (shares * price) / DEFAULT_DECIMALS_FACTOR;
        DepositRecord memory depositRecord = _getDepositRecord(lender);
        if (assets > depositRecord.principal) {
            uint256 yield = assets - depositRecord.principal;
            tranchesAssets[trancheIndex] -= uint96(yield);
            // Round up the number of shares the lender has to burn in order to receive
            // the given amount of yield. Round-up applies the favor-the-pool principle.
            shares = Math.ceilDiv(yield * DEFAULT_DECIMALS_FACTOR, price);
            ERC20Upgradeable._burn(lender, shares);
            poolSafe.withdraw(lender, yield);
            emit YieldPaidOut(lender, yield, shares);
        }
    }
    poolSafe.resetUnprocessedProfit();
    pool.updateTranchesAssets(tranchesAssets);
}
```

Recommendation: For the sake of smoothness, implementing a try/catch statement for the `poolSafe.withdraw()` call is worth adding.

Huma: `try/catch` has been added in PR 405. We also fixed the same issue in the EA replacement function as discussed in the past. We also discovered by ourselves that `FirstLossCover` has the same problem, so we fixed that as well.

Spearbit: Fixed.

5.3.20 An epoch can be closed before processing a tranche's unprocessed yield, leading to less optimal share redemptions

Severity: Low Risk

Context: TrancheVault.sol#L421-L444, EpochManager.sol#L94-L141

Description: The autotask peforms the following tasks regularly:

- Fetch all pools where the current epoch `endTime` is in the past.
- Call `processYieldForLenders()` on the junior and senior tranche.
- Call `closeEpoch()` on the epoch manager contract.
- Call `investFeesInFirstLossCover()` on the pool fee manager contract.
- Call `payoutYield()` on all first loss cover contracts of the pool.

While `processYieldForLenders()` can be called at any time to process any tranche profit and ultimately redeem any yield earned by users who have opted not to reinvest their earnings, there is no guarantee that `closeEpoch()` is not called before the autotask is able to. As a result, there may be some unprocessed tranche profit which is not accounted for, leading to inefficient epoch processing and less optimal share redemptions.

Recommendation: Add a hook call to `processYieldForLenders()` into the `closeEpoch()` function to ensure optimal behaviour at all times.

Huma: It's quite complicated to call `processYieldForLenders()` in `_processRedemptionRequests()` since it requires interface changes. So let's simply add a check to block redemption request processing if there is unprocessed profit. Fixed in PR 449.

Spearbit: Fixed.

5.3.21 Edge case for totalAssets can cause divide by zero error

Severity: Low Risk

Context: TrancheVault.sol#L608

Description: A borrower can default, causing a decrease in `totalAssets` (`pool.tranchesAssets(index)`), without a corresponding decrease in `totalSupply`. This scenario may lead to a situation where the `totalSupply` of the vault is non-zero while `totalAssets` is zero. In such a case, no one will be able to deposit, as the `_convertToShares` function will revert with a divide-by-zero error.

Recommendation: A way to handle it is to return 0 if `totalAssets` are zero when calculating shares from amount.

Huma: Fixed in PR 388.

Spearbit: Fixed.

5.3.22 Inconsistent behavior in `makePrincipalPaymentAndDrawdownWithReceivable()`

Severity: Low Risk

Context: ReceivableBackedCreditLine.sol#L181-L184

Description: Calling `makePrincipalPaymentAndDrawdownWithReceivable()` doesn't update the borrower's credit record and due info correctly when `paymentAmount == drawdownAmount`, unlike calling `makePrincipalPaymentWithReceivable()` and `drawdownWithReceivable()` separately.

Recommendation: Use `_makePrincipalPayment()` and `_drawdown()` separately to ensure accurate updates to the credit record and due information.

Huma: The difference in behavior is intentional. This function is intended for a very specific use case for one of our partners, where they would make principal payment and drawdown of equal amounts `D` every weekday (except the initial drawdown to kick start the credit line). This implementation is deliberately chosen to circumvent the necessity of imposing additional yield charges on each transaction. Ordinarily, using `makePrincipalPaymentWithReceivable()` and `drawdownWithReceivable()` separately would result in extra yield charges for the borrower. However, this implementation ensures that yield is charged only once for the borrowed amount `D`, rather than multiple times.

You are correct in that `CreditRecord` and `DueDetail` would not be guaranteed to be updated. In most cases this is fine - there is usually nothing to update since the total outstanding principal doesn't change and no yield or fee is getting paid. The only scenarios requiring updates to these structures are when a new bill needs to be generated, and we expect that to be handled by either the regular `makePaymentWithReceivable()` function, the `refreshCredit()` process (which is called periodically by one of our Autotasks), or in cases where the principal payment amount differs from the drawdown amount. There are some edge cases where the next bill should be generated here when the two amounts are equal, e.g. somehow `refreshCredit()` is down for more than a month, which is very unlikely to happen. We'll consider updating the two structs here if needed, but it's relatively low priority and low impact.

Spearbit: We agree it's intentional behavior. Acknowledged.

5.3.23 Unintended unpausing credit for borrowers

Severity: Low Risk

Context: Credit.sol#L418

Description: There's an issue where a borrower, upon having their credit `Paused`, can resume their credit status to `GoodStanding` by making a full payment off `payoffAmount`. Unpausing the credit allows the borrower to have access to further drawdowns, breaking the pausing functionality

Recommendation: It is recommended to add a check to ensure a borrower's credit isn't `Paused` before processing any payments, or don't update for `GoodStanding` when the state is `Paused`.

Huma: As discussed, we will remove `pause()` and `unpause()` feature. When we need to pause a credit, the EA will just set the `creditLimit` to 0. The pause/unpause functionality is removed in commit a62b1555.

Spearbit: Verified.

5.3.24 Credit can be reapproved when paused

Severity: Low Risk

Context: CreditManager.sol#L226

Description: Credit is approved by the EA service account which is ultimately delegated by the pool owner. The EA service account may have an account paused and then proceed to reapprove credit for the same account, overwriting sensitive loan data that should otherwise remain untouched until the loan has reached the end of its lifetime.

Note: this action would assume some degree of negligence by the EA. Under normal protocol actions, this should not happen.

Recommendation: Add an explicit check for this alongside the `cr.state > CreditState.Approved` check. It seems useful to allow for credit to be repproved for a borrower when the loan has been paid off and is in the `CreditState.Deleted` state.

Huma: We are removing the pause/unpause functionality in PR 407.

Spearbit: Fixed.

5.3.25 Unsynchronized `PoolConfig` state variables in interdependent contracts

Severity: Low Risk

Context: PoolConfig.sol#L450-L469

Description: In the `PoolConfig` contract, updates to state variables aren't automatically reflected in other dependent contracts. This results in a mismatch, as other contracts continue using outdated values, potentially leading to functional discrepancies and security concerns.

Recommendation: To ensure consistency, it's important to update dependent contracts whenever a state variable in `PoolConfig` is changed.

Huma: We did consider automatically updating all downstream contracts that cache the addresses to the new value, but decided against that due to scalability concerns: `PoolConfig` would have to know about which contracts had cached the value and update them one by one. What's worse, it'll not only have to do this for `FirstLossCovers`, but also for all the contract addresses that are cached by other contracts. This means that the majority of the setters, if not all, would need to know about their dependencies, and `PoolConfig` would be bloated. So we've decided that it's the PoolOwner's responsibility to update all other contracts' caches by calling `updatePoolConfigData` when something changes in `PoolConfig`.

Spearbit: Acknowledged.

5.3.26 Missing credit limit check in `_updateLimitAndCommitment`

Severity: Low Risk

Context: CreditManager.sol#L491-L539

Description: There are a number constraints that are enforced when the pool owner approves credit for a borrower. However, one of these constraints is not enforced when updating credit limits during a loan's term. One of these includes the `poolConfig.getPoolSettings().maxCreditLine >= creditLimit` check which ensures no loan credit line exceeds the pool configuration parameters.

While `_approveCredit()` does not allow for setting the initial `creditLimit` and `committedAmount` values to zero, there seems to be some legitimate use cases to allow for this within `_updateLimitAndCommitment()` function. i.e. the pool owner can prevent the borrower from performing any further drawdown. Alternatively, the required committed amount may also be removed, meaning the borrower is not expected to pay yield on any unused funds.

Recommendation: The following check seen in `_approveCredit()` can also be added to the `_updateLimitAndCommitment()` function.

```
PoolSettings memory ps = poolConfig.getPoolSettings();
if (creditLimit > ps.maxCreditLine) {
    revert Errors.CreditLimitTooHigh();
}
```

It might be worth also adding the `committedAmount > creditLimit` invariant check to the internal function instead of `CreditLineManager.updateLimitAndCommitment()`.

Huma: Fixed in PR 414. Also moved the check into the internal function as suggested in past discussions.

Spearbit: Fixed.

5.3.27 Excess yield paid may be lost when updating a loan's yield

Severity: Low Risk

Context: CreditManager.sol#L426-L479

Description: During a loan's lifetime, the EA service account may update the yield charged on the borrowed funds. To avoid retroactively updating any previous accrued but not yet paid interest, the `_updateYield()` function will recompute yield due by taking into consideration how many days are remaining in the current billing period.

If the yield rate is decreased during a billing period, then the new `dd.accrued` amount will have also decreased. If the borrower has already paid the yield owed in the current billing period, then `dd.paid > dd.accrued` may hold true. The protocol attempts to recalculate the yield due in the below snippet of code. Yield rate adjustments can happen during a billing cycle.

```
function recomputeYieldDue(
    uint256 nextDueDate,
    uint256 oldYieldDue,
    uint256 oldYieldInBps,
    uint256 newYieldInBps,
    uint256 principal
) external view returns (uint256 updatedYield) {
    uint256 daysRemaining = calendar.getDaysRemainingInPeriod(nextDueDate);
    // Note that we do not divide by `(HUNDRED_PERCENT_IN_BPS * DAYS_IN_A_YEAR)` here since division
↪   rounds down.
    // We will do summation before division at the end for better precision.
    uint256 newYieldDueForDaysRemaining = principal * newYieldInBps * (daysRemaining - 1);
    uint256 oldYieldDueForDaysRemaining = principal * oldYieldInBps * (daysRemaining - 1);
    return
        (oldYieldDue *
            HUNDRED_PERCENT_IN_BPS *
            DAYS_IN_A_YEAR +
            newYieldDueForDaysRemaining -
            oldYieldDueForDaysRemaining) / (HUNDRED_PERCENT_IN_BPS * DAYS_IN_A_YEAR);
}
```

Consequently, there is some excess yield paid that is not accounted for and remains locked within the protocol.

Recommendation: Consider allowing only for the EA service account to make yield rate updates at the end of a billing period. However, the implementation considers yield updates during a billing period because `recomputeYieldDue()` explicitly handles this adjustment. In this case, if `dd.paid > dd.accrued` then attribute the yield to any other unpaid component of the borrower's loan.

Huma: This is actually work as intended. Once a payment for a period is paid, it is final and we do not change it. Also, we expect the pool owners only make yield rate changes at the beginning of each period when nobody has made payment for this new period. Thus, in reality, the chance for the described scenario to happen is extremely low.

We do think it is better to make the new interest rate effective starting from the next period as much as we can no matter it is an increase or decrease. It will lead to a better user experience anyway. In a Web2 world, we will record

the new rate and specify on which date the new rate takes effect. We do not want to introduce such complexity to the smart contract, however, with a small change, we can achieve the vast majority of the desired outcome.

We plan not to update the yield due when the yieldInBps is updated no matter the yieldInBps has increased or decreased. Instead, we wait until the next time when the system refreshes the yield due for the borrower to apply the new rate. The yield due is refreshed under three situations:

1) Auto payment of the yield due.
2) At the end of late payment grace period.
3) The borrower has made additional drawdown of principals.

Both 1) and 2) happens at the first few days of a period. If the new yieldInBps takes effect when this works, it is perfect. 3) is very uncommon for business lending. We will document it to let the borrower know if they make additional drawdown in a period, it will trigger any pending yield rate change to take effect.

We do not think this is a big change, basically, we do not update yield due when yieldInBps is updated. Fixed in PR 435.

Spearbit: Fixed.

5.3.28 Implement boundary checks in setter functions

Severity: Low Risk

Context: PoolConfig.sol#L457, PoolConfig.sol#L499, PoolConfig.sol#L520, PoolConfig.sol#L534, PoolConfig.sol#L545, CreditManager.sol#L233

Description: The setter functions in the contract are designed for use by trusted accounts. However, there's still a risk of making incorrect updates due to human error or oversight. To mitigate this, it's essential to implement boundary checks for each setter function across all context files.

Recommendation: It is recommended to implement boundary checks for all functions mentioned in the context.

Huma: Done in this PR 393. Update: we removed the check on `yieldInBps` since it's valid for it to exceed 10000.

Spearbit: Fixed.

5.4 Gas Optimization

5.4.1 Redundant `_onlyDeployer` check

Severity: Gas Optimization

Context: PoolFactory.sol#L742

Description: Upon deploying new pools, an internal call is made to `_createPoolContracts()` from `deployPool()`. However, there are some duplicate checks made which can be removed.

Recommendation: Remove the `_onlyDeployer(msg.sender)` check in `_createPoolContracts()`.

Huma: Fixed in PR 419.

Spearbit: Verified fix.

5.4.2 Pool config read can be optimised by moving it within an `if` statement

Severity: Gas Optimization

Context: EpochManager.sol#L236-L246

Description: The `_processEpoch()` function will attempt to process junior tranche token redemption requests if the total requested amount is non-zero. The amount which can be claimed is enforced by the `maxSeniorJuniorRatio` pool config setting. This call can be optimised by moving it within the `if` statement for where it is known that junior tranche shares are being requested.

Recommendation: Making the following adjustment to `_processEpoch()`:

```
if (juniorSummary.totalSharesRequested > 0) {
    LPConfig memory lpConfig = poolConfig.getLPConfig();
    uint256 maxSeniorJuniorRatio = lpConfig.maxSeniorJuniorRatio;
    availableAmount = _processJuniorRedemptionRequests(
        tranchesAssets,
        juniorPrice,
        maxSeniorJuniorRatio,
        juniorSummary,
        availableAmount
    );
}
```

Huma: Fixed in PR 383. Also made some readability improvements.

Spearbit: Fixed.

5.4.3 In some endpoints all instances of `borrower` can be replaced with `msg.sender` after the check

Severity: Gas Optimization

Context: ReceivableBackedCreditLine.sol#L78-L99, ReceivableBackedCreditLine.sol#L147-L210, CreditLine.sol#L16-L26, ReceivableFactoringCredit.sol#L59-L81, ReceivableFactoringCredit.sol#L84-L102

Description: In the endpoints in this context, it is checked that the `borrower` should be the same as the `msg.sender`.

Recommendation: In the endpoints in this context, all instances of `borrower` can be replaced with `msg.sender` after the check.

Also one can:

1. Remove the `borrower` input parameter.
2. Remove the check.
3. Replace all instances of the `borrower` with `msg.sender`.
4. Comments can be added to indicate that the `msg.sender` is the `borrower`.

Huma: We removed the redundant borrower params in PR 459.

Spearbit: Fixed.

5.4.4 Calculating the `numMonths` variable in `getDaysRemainingInPeriod` can be simplified

Severity: Gas Optimization

Context: Calendar.sol#L26-L27

Description: `DTL.diffMonths` ignores the `day` portion of the Gregorian date so one can feed `block.timestamp` directly into `DTL.diffMonths`(unless there are some issues with the Julian day to/from Gregorian date conversions in the library).

Recommendation:

```
- uint256 startDateOfMonth = _getStartOfMonth(block.timestamp);
- uint256 numMonths = DTL.diffMonths(startDateOfMonth, endDate);
+ uint256 numMonths = DTL.diffMonths(block.timestamp, endDate);
```

Also note that the tests all pass with this change:

```
yarn hardhat test --network hardhat --grep "Calendar Test"
```

Huma: Fixed in PR 404.

Spearbit: Fixed.

5.4.5 One of the input parameters to `_computeYieldNextDue` can be simplified

Severity: Gas Optimization

Context: CreditDueManager.sol#L143, CreditDueManager.sol#L78, CreditDueManager.sol#L132

Description: The following expression can be simplified:

```
cr.unbilledPrincipal + cr.nextDue - cr.yieldDue + dd.principalPastDue,
```

We have:

```
newDD = _deepCopyDueDetail(dd);
// ...
if (isLate) {
    if (timestamp > cr.nextDueDate) {
        // ...
        newDD.principalPastDue += cr.nextDue - cr.yieldDue;
        // ...
             cr.unbilledPrincipal + cr.nextDue - cr.yieldDue + dd.principalPastDue,
// ...
```

and so:

```
newDD.principalPastDue == cr.nextDue - cr.yieldDue + dd.principalPastDue
```

Recommendation: The simplified form would be:

```
cr.unbilledPrincipal + newDD.principalPastDue,
```

Huma: While it's true that the two are equivalent, we think it's more obvious from the first implementation that we are calculating the total outstanding principal here. Since the gas saving is negligible, we are inclined to keep it as-is.

Spearbit: Acknowledged.

5.4.6 `latePaymentDeadline` can be deferred in `getNextBillRefreshDate`

Severity: Gas Optimization

Context: CreditDueManager.sol#L320-L333

Description: `latePaymentDeadline` in `getNextBillRefreshDate` needs to be only calculated when:

```
cr.state == CreditState.GoodStanding && cr.nextDue != 0
```

is true.

Recommendation: Move the calculation of `latePaymentDeadline` inside the `if` block in this context:

```
function getNextBillRefreshDate(
    CreditRecord memory cr
) public view returns (uint256 refreshDate) {
    if (cr.state == CreditState.GoodStanding && cr.nextDue != 0) {
        // If this is the first time ever that the bill has surpassed the due date and the bill has
↪   unpaid amounts,
        // then we don't want to refresh the bill since we want the user to focus on paying off the
↪   current due.
        PoolSettings memory poolSettings = poolConfig.getPoolSettings();
        uint256 latePaymentDeadline = cr.nextDueDate +
            poolSettings.latePaymentGracePeriodInDays *
            SECONDS_IN_A_DAY;
        return latePaymentDeadline;
    }
    return cr.nextDueDate;
}
```

Huma: Good catch. Addressed in PR 428.

Spearbit: Fixed.

5.4.7 `PoolConfig(pools[_poolId].poolConfigAddress)` can be cached

Severity: Gas Optimization

Context: PoolFactory.sol#L492-L516

Description/Recommendation: `PoolConfig(pools[_poolId].poolConfigAddress)` can be cached in this context.

Huma: Fixed in PR 421.

Spearbit: Fixed.

5.4.8 `_onlySystemMoneyMover` makes multiple calls to `poolConfig` to query different contract addresses.

Severity: Gas Optimization

Context: PoolSafe.sol#L118-L122, PoolSafe.sol#L61, PoolSafe.sol#L54, PoolSafe.sol#L76-L77

Description: `_onlySystemMoneyMover` makes multiple calls to `poolConfig` to query different contract addresses:

```
function _onlySystemMoneyMover(address account) internal view {
    if (
        account != poolConfig.seniorTranche() &&
        account != poolConfig.juniorTranche() &&
        account != poolConfig.credit() &&
        account != poolConfig.poolFeeManager() &&
        !poolConfig.isFirstLossCover(account)
    ) revert Errors.AuthorizedContractCallerRequired();
}
```

`resetUnprocessedProfit`, `addUnprocessedProfit`, `getAvailableBalanceForPool` also makes multiple calls to `poolConfig`.

Recommendation: Combine all these calls into just one call to `poolConfig` to query and check the addresses.

Huma: We originally placed the `_onlySystemMoneyMover()` function, along with some other permission validation functions, in the `PoolConfig` contract. However, due to contract size limitations—`PoolConfig` once exceeded the 24KB limit—we had to relocate them. Currently, `PoolConfig` is approximately 20KB, leaving limited space for expansion. We need to assess the impact on the contract's size if we reintegrate this function and others you mentioned below. We would like to preserve sufficient capacity for potential future pool configuration options. Should re-adding these functions cause `PoolConfig` to exceed its size constraints, an alternative solution would be to read the `poolConfig` storage variable into memory first before calling functions on it, offering some gas savings. Cached more addresses across multiple contracts in PR 451.

Spearbit: Fixed.

5.4.9 `HUNDRED_PERCENT_IN_BPS - lpConfig.tranchesRiskAdjustmentInBps` can be stored in the storage instead of `lpConfig.tranchesRiskAdjustmentInBps`

Severity: Gas Optimization

Context: RiskAdjustedTranchesPolicy.sol#L32

Description: Only spot that `lpConfig.tranchesRiskAdjustmentInBps` is used it might be cheaper to store `(HUNDRED_PERCENT_IN_BPS - lpConfig.tranchesRiskAdjustmentInBps)` in the storage instead.

Recommendation: `HUNDRED_PERCENT_IN_BPS - lpConfig.tranchesRiskAdjustmentInBps` can be stored in the storage instead of `lpConfig.tranchesRiskAdjustmentInBps`:

```
diff --git a/contracts/common/PoolConfig.sol b/contracts/common/PoolConfig.sol
index d33462a..52b8fd2 100644
--- a/contracts/common/PoolConfig.sol
+++ b/contracts/common/PoolConfig.sol
@@ -54,7 +54,7 @@ struct LPConfig {
     uint8 maxSeniorJuniorRatio;
     // The fixed yield for senior tranche. Either this or tranchesRiskAdjustmentInBps is non-zero
     uint16 fixedSeniorYieldInBps;
-    // Percentage of yield to be shifted from senior to junior. Either this or fixedSeniorYieldInBps
↪  is non-zero
+    // Percentage of yield to be kept for the senior tranche (the rest is shifted to junior). Either
↪  this or fixedSeniorYieldInBps is non-zero
     uint16 tranchesRiskAdjustmentInBps;
     // How long a lender has to wait after the last deposit before they can withdraw
     uint16 withdrawalLockoutPeriodInDays;
diff --git a/contracts/liquidity/RiskAdjustedTranchesPolicy.sol
↪  b/contracts/liquidity/RiskAdjustedTranchesPolicy.sol
index c85e81e..0ca0aad 100644
--- a/contracts/liquidity/RiskAdjustedTranchesPolicy.sol
+++ b/contracts/liquidity/RiskAdjustedTranchesPolicy.sol
@@ -23,13 +23,10 @@ contract RiskAdjustedTranchesPolicy is BaseTranchesPolicy {

         LPConfig memory lpConfig = poolConfig.getLPConfig();
         // If we disregard rounding errors, the following calculation is mathematically equivalent to:
-        // seniorProfit = profit * seniorAssets / (seniorAssets + juniorAssets)
-        // seniorProfit -= seniorProfit * lpConfig.tranchesRiskAdjustmentInBps / HUNDRED_PERCENT_IN_BPS
+        // seniorProfit = profit * seniorAssets * lpConfig.tranchesRiskAdjustmentInBps
+        // seniorProfit /= (HUNDRED_PERCENT_IN_BPS * (seniorAssets + juniorAssets))
         // The two steps are combined into one to minimize rounding errors due to integer division.
-        seniorProfit =
-            (profit *
-                seniorAssets *
-                (HUNDRED_PERCENT_IN_BPS - lpConfig.tranchesRiskAdjustmentInBps)) /
+        seniorProfit = (profit * seniorAssets * lpConfig.tranchesRiskAdjustmentInBps) /
             (HUNDRED_PERCENT_IN_BPS * (seniorAssets + juniorAssets));
         remainingProfit = profit - seniorProfit;
```

Huma: The amount of gas saved seems minimal. More importantly, from a business perspective, it's easier to explain what `lpConfig.tranchesRiskAdjustmentInBps` is than the opposite value, so we will keep the stored value as-is.

Spearbit: Acknowledged.

5.4.10 Redundant `address(0)` checks in contract creation in `PoolFactory`

Severity: Gas Optimization

Context: PoolFactory.sol#L648, PoolFactory.sol#L658, PoolFactory.sol#L665, PoolFactory.sol#L672, PoolFactory.sol#L679, PoolFactory.sol#L686, PoolFactory.sol#L693, PoolFactory.sol#L700, PoolFactory.sol#L707, PoolFactory.sol#L714, PoolFactory.sol#L721

Description: The deployment of proxy contracts contains a double-check for address(0), these additional checks increase gas consumption and add to the bytecode without providing extra security or functionality.

Recommendation: It's recommended to remove the redundant address(0) checks. This can reduce gas costs and simplify the code. For example:

```
  function addReceivable() external {
    _onlyDeployer(msg.sender);
-   if (receivableImpl == address(0)) revert Errors.ZeroAddressProvided();
    address receivable = _addProxy(receivableImpl);
    emit ReceivableCreated(receivable);
  }
```

Huma: Removed all redundant check in PR 419.

Spearbit: Fixed.

5.4.11 Avoid roundtrip in `EpochManager` when `tranche.currentRedemptionSummary` is used

Severity: Gas Optimization

Context: EpochManager.sol#L79-L86, EpochManager.sol#L110-L111, TrancheVault.sol#L447-L454

Description: `tranche.currentRedemptionSummary()` is only used in the `EpochManager` contract and in the `TrancheVault` it calls back to `EpochManager` to get the current epoch id.

Recommendation: To avoid this round trip it would be best to pass the current epoch id instead to this endpoint and avoid the extra call back to the `EpochManager`:

```
// file: EpochManager.sol

tranche.currentRedemptionSummary(_currentEpoch.id)
```

Huma: Addressed in PR 385.

Spearbit: Fixed.

5.5 Informational

5.5.1 `_daysFromDate`'s and `_daysToDate`'s implementations would need to be verified and tested

Severity: Informational

Context: BokkyPooBahsDateTimeLibrary.sol#L43-L73, BokkyPooBahsDateTimeLibrary.sol#L75-L109

Description: Due to lack of time the integrity of the implementation of the functions in this context are not verified:

- `_daysFromDate`
- `_daysToDate`

Recommendation: It is recommend to run a differential test suit to verify that these `2` functions return the expected results given inputs in the range of `1970/01/01` to X years in the future (can be high enough for confidence maybe 100 or 1000 years).

The alternative implementations that one would test against can be the simpler but more inefficient versions of the functions in this context.

5.5.2 `getDaysDiff` has a constraint which is not enforced

Severity: Informational

Context: Calendar.sol#L74-L79

Description: It is possible that `startDate > endDate` holds true if `block.timestamp > endDate && startDate == 0`. This should not be something that is allowed.

Recommendation: Re-order the first two checks of `getDaysDiff()` so the constraints are properly enforced.

Huma: Fixed in PR 404.

Spearbit: Verified fix.

5.5.3 Evaluation agent can avoid being removed from `PoolConfig`

Severity: Informational

Context: PoolConfig.sol#L342

Description: A pool's evaluation agent is tasked with underwriting risk associated with the pool. The pool owner or huma master admin can set a new evaluation agent by calling `setEvaluationAgent()`. If the old EA account has withdrawable funds, then `withdrawEAFee()` will be called. If this account can intentionally cause this to revert, then they can also avoid being removed altogether.

Recommendation: Consider implementing a try/catch statement here to ensure that new EA accounts can always be set.

Huma: Fixed in PR 405.

Spearbit: Verified fix.

5.5.4 Potential underflow in `seniorAvailableCap` when a junior tranche defaults on a loan

Severity: Informational

Context: Pool.sol#L210-L213

Description: The `seniorAvailableCap` calculation will potentially revert by underflowing if the junior tranche experiences any loan default that causes `assets[JUNIOR_TRANCHE] * config.maxSeniorJuniorRatio` to fall below the amount of assets held by the senior tranche.

```
uint256 seniorAvailableCap = assets[JUNIOR_TRANCHE] *
    config.maxSeniorJuniorRatio -
    assets[SENIOR_TRANCHE];
```

Recommendation: Fortunately, this only causes senior tranche deposits to fail until ratio is reached again. However, it is worth explicitly handling this silently or via a custom revert error.

Huma: Fixed in PR 396.

Spearbit: Verified fix.

5.5.5 Pool config account naming can be improved

Severity: Informational

Context: PoolConfig.sol#L692-L702

Description: The naming conventions for the pool config admin accounts can be confusing and do not necessarily clearly outline the implementation behaviour.

Recommendation: Consider renaming `onlyOwnerOrHumaMasterAdmin()` to `onlyPoolOwnerOrHumaOwner()` and `onlyHumaMasterAdmin()` to `onlyHumaOwner()`.

Huma: Fixed in PR 391.

Spearbit: Fixed.

5.5.6 Improper permissioning of `withdrawProtocolFee`

Severity: Informational

Context: PoolFeeManager.sol#L107-L125

Description: The protocol fee can be withdrawn by the huma config owner. This account is permissioned to make any configuration change to the protocol and should not be used for other purposes such as claiming a protocol fee.

Recommendation: Update this such that only `humaConfig.humaTreasury()` account can call `withdrawProtocolFee()` and not `humaConfig.owner()`.

Huma: Fixed in PR 390.

Spearbit: Verified fix.

5.5.7 `coveredLoss` parameter shadows storage variable

Severity: Informational

Context: FirstLossCover.sol#L421

Description: The variable naming of the `coveredLoss` parameter in the `_calcLossRecovery()` function shadows an existing storage variable.

Recommendation: Consider renaming this parameter.

Huma: Fixed in PR 441.

Spearbit: Verified fix.

5.5.8 `recoverLoss` and `coverLoss` functions can be simplified

Severity: Informational

Context: FirstLossCover.sol#L192-L225

Description: These functions add some unnecessary steps to a simple calculation.

Recommendation: The following simplifications can be made:

```
  function coverLoss(uint256 loss) external returns (uint256 remainingLoss) {
      poolConfig.onlyPool(msg.sender);

      uint256 coveredAmount;
      (remainingLoss, coveredAmount) = _calcLossCover(loss);

      if (coveredAmount > 0) {
          poolSafe.deposit(address(this), coveredAmount);

-         uint256 newCoveredLoss = coveredLoss;
-         newCoveredLoss += coveredAmount;
+         coveredLoss += coveredAmount;
-         coveredLoss = newCoveredLoss;

          emit LossCovered(coveredAmount, remainingLoss, newCoveredLoss);
      }
  }

  function recoverLoss(uint256 recovery) external returns (uint256 remainingRecovery) {
      poolConfig.onlyPool(msg.sender);

      uint256 recoveredAmount;
      (remainingRecovery, recoveredAmount) = _calcLossRecovery(coveredLoss, recovery);

      if (recoveredAmount > 0) {
-         uint256 currCoveredLoss = coveredLoss;
-         currCoveredLoss -= recoveredAmount;
+         coveredLoss -= recoveredAmount;
-         coveredLoss = currCoveredLoss;

          poolSafe.withdraw(address(this), recoveredAmount);

          emit LossRecovered(recoveredAmount, currCoveredLoss);
      }
  }
```

Huma: This is indeed for gas savings. We are trying to follow a pattern of:

1. Reading a storage variable into memory;
2. Updating the copy;
3. Writing it back to storage.

You are right that we can simplify as you suggested. We still used this pattern here just in case we need to do more to `coveredLoss` than just a simple subtraction in the future. You are also right that we should cache `coveredLoss` before calling `_calcLossRecovery()`, so we updated that in PR 439.

Spearbit: Agree with the resolution, considering it as fixed.

5.5.9 Inconsistency of allowed callers to `makePrincipalPayment...`

Severity: Informational

Context: ReceivableBackedCreditLine.sol#L147-L155, CreditLine.sol#L44-L49

Description: `_onlySentinelServiceAccount()` can call `makePrincipalPayment` endpoint in `CreditLine`. This is in contrast to `makePrincipalPaymentWithReceivable` in `ReceivableBackedCreditLine` where only the `borrower` can call into.

Recommendation: It would be best to have consistency across the different credit types has which entities can call `makePrincipalPayment...`.

Huma: Since the Sentinel Service account is used for Autopay and it is enabled for `makePayment()` already, we don't see a need to enable it for `makePrincipalPayment()`. So we will remove its access in `CreditLine`. Fixed in PR 438.

Spearbit: Fixed.

5.5.10 One can call `declarePayment` to set `receivableInfo.paidAmount` to any value below the set cap

Severity: Informational

Context: Receivable.sol#L105

Description: There is no on-chain payment associated to this endpoint and the token owner can set the `receivableInfo.paidAmount` to any desired value capped to `receivableInfo.receivableAmount`. `receivableInfo.paidAmount` is not even used in the protocol codebase.

Recommendation: More comments should be added for this endpoint for example to indicate whether this is solely used for off-chain analysis and monitoring.

Huma: When the borrower makes a payment by calling `makePaymentWithReceivable()`, they may or may not have received payments from the debtor of the receivable (i.e. the party that owes the borrower money in the receivable transaction), so the borrower may have made payment with their own money instead of the money received from the debtor. `paidAmount` should only be updated when they received the payment from the debtor, hence we are tying it with credit line payment and it's the borrower's responsibility to keep the value up-to-date. Added comments to PR 450.

Spearbit: Fixed.

5.5.11 `makePrincipalPaymentAndDrawdownWithReceivable` makes `msg.sender` send and receive the same amount of underlying token to the `poolSafe` and back

Severity: Informational

Context: ReceivableBackedCreditLine.sol#L181-L195, ReceivableBackedCreditLine.sol#L155

Description: `borrower` is checked to be the `msg.sender`.

These two lines would transfer the same amount of the underlying token from the `msg.sender` to the pool safe and back.

```
poolSafe.deposit(msg.sender, AMOUNT);
poolSafe.withdraw(borrower, AMOUNT);
```

Recommendation: The roundtrips can be removed.

Huma: Correct, and that's the intended behavior. When the payment amount and drawdown amount are the same, we don't want to call `makePaymentWithReceivable()` and then `drawdownWithReceivable()` because that would cause additional yield to be generated, but we do like to show that the pool has received payment and then disbursed the same amount back to the borrower, hence the `deposit` and `withdraw` roundtrip here. We can probably add some comments to make our intention clear here.

Spearbit: In that case documenting why this behaviour is required would be great.

Huma: Addressed in PR 437.

Spearbit: Fixed.

5.5.12 Setter functions for `creditDueManager` and `creditManager` are missing

Severity: Informational

Context: PoolConfig.sol#L114, PoolConfig.sol#L116

Description: Setter functions for `creditDueManager` and `creditManager` are missing.

Recommendation: If required implement setter functions for these addresses in `PoolConfig`.

Huma: Fixed in PR 413.

Spearbit: Fixed.

5.5.13 `poolAdmins` storage parameter can be removed from `HumaConfig`

Severity: Informational

Context: HumaConfig.sol#L37

Description: `poolAdmins` and related functions are not used.

Recommendation: If this storage parameter is not going to be used in the future it might be best to remove all the code related to this parameter.

Huma: Fixed in PR 430.

Spearbit: Fixed.

5.5.14 The return expression in `getDaysDiff` can be simplified

Severity: Informational

Context: Calendar.sol#L95

Description/Recommendation: The expression in this context can be simplified:

```
return numMonthsPassed * DAYS_IN_A_MONTH + endDay - startDay;
```

Huma: Fixed in PR 404.

Spearbit: Fixed.

5.5.15 The check to set `humaTreasury` is different than the other endpoints

Severity: Informational

Context: HumaConfig.sol#L235-L238

Description: When one sets the `humaTreasury` the new and old values are compared and if they are not equal `humaTreasury` is updated. This differs from other endpoints where the comparison is not performed for example `eaServiceAccount`, `eaNFTContractAddress`.

Recommendation: For consistency it might be best to keep the implementation of these setter functions similar.

Huma: Fixed in PR 402.

Spearbit: Fixed.

5.5.16 Make sure `safeTransferFrom` and `transferFrom` in `EvaluationAgentNFT` would revert

Severity: Informational

Context: EvaluationAgentNFT.sol#L57-L80

Description: Other contracts in the codebase when one wants to transfer tokens, the implementation would cause a revert. Reverting might be a better implementation compared to a silent noop execution since the calling user/party might assume that the transfer had happened successfully.

Recommendation: Make sure `safeTransferFrom` and `transferFrom` in `EvaluationAgentNFT` would revert.

Huma: Fixed in PR 401.

Spearbit: Fixed.

5.5.17 Conditional statement in `_checkDrawdownEligibility` can be simplified

Severity: Informational

Context: Credit.sol#L571

Description: In this context we have:

```
if (cr.nextDueDate > 0 && block.timestamp < cr.nextDueDate)
    revert Errors.FirstDrawdownTooEarly();
```

`block.timestamp < cr.nextDueDate` being `true` implies that `cr.nextDueDate > 0`.

Recommendation: The conditional statement can be simplified to:

```
if (block.timestamp < cr.nextDueDate) ...
```

Huma: Fixed in PR 399.

Spearbit: Fixed.

5.5.18 Simpler expression to update `cr.nextDue` in `_makePrincipalPayment`

Severity: Informational

Context: Credit.sol#L498

Description: In this context we have:

```
uint256 principalDue = cr.nextDue - cr.yieldDue;
// ...
uint256 principalDuePaid;
// ...
} else {
    principalDuePaid = principalDue;
    // ...
    cr.nextDue = uint96(cr.nextDue - principalDuePaid);
```

Recommendation: The expression for `cr.nextDue` in `_makePrincipalPayment` can be simplified to:

```
cr.nextDue = cr.yieldDue;
```

Huma: Fixed in PR 399.

Spearbit: Fixed.

5.5.19 Unused elements of the `ReceivableState` enum

Severity: Informational

Context: CreditStructs.sol#L92-L101

Description: Only `Minted`, `Approved`, `PartiallyPaid`, and `Paid` elements of `ReceivableState` are used.

Recommendation: Delete the unused elements or document why they currently exist in this `enum`.

Huma: While some values are not currently used, we will keep these states for future expansion. Re: `Deleted`, it doesn't make much sense to mark a receivable as `Deleted` when it's burned since it won't even exist. We'll keep the enum value though so that the receivable owner has the flexibility to keep unused receivables around and mark them as `Deleted` instead of burning them. As discussed, Huma will keep the values as-is for future expansion purposes.

Spearbit: Acknowledged.

5.5.20 Credit limit invariant should be checked within `_updateLimitAndCommitment`

Severity: Informational

Context: CreditLineManager.sol#L168-L170, CreditManager.sol#L491

Description:

```
if (committedAmount > creditLimit) revert Errors.CommittedAmountGreaterThanCreditLimit();
_updateLimitAndCommitment(getCreditHash(borrower), creditLimit, committedAmount);
```

The above check should happen atomically within the update flow and not just to this specific endpoint of `CreditLineManager.updateLimitAndCommitment`.

Recommendation: It might be best to move the `if` block inside the internal function `_updateLimitAndCommitment` to guarantee that the invariant in question if always checked even if one starts using `_updateLimitAndCommitment` in another flow.

Huma: Fixed in PR 414.

Spearbit: Fixed.

5.5.21 Use underscore prefix for internal functions across the codebase

Severity: Informational

Context: CreditLine.sol#L72

Description: In the context above the internal function names do not start with an underscore _.

Recommendation: Use underscore prefix for internal functions across the codebase:

```
function _functionName(/*...*/) internal //...
```

Huma: We decided to make it a public function instead so clients (dApp, Autotask, SDK etc.) can also call it PR 398.

Spearbit: Fixed.

5.5.22 Use nested `if` blocks

Severity: Informational

Context: CreditManager.sol#L208-L215

Description: In `_approveCredit` we have:

```
if (committedAmount == 0 && designatedStartDate != 0)
    revert Errors.CreditWithoutCommitmentShouldHaveNoDesignatedStartDate();
if (designatedStartDate > 0 && block.timestamp > designatedStartDate)
    revert Errors.DesignatedStartDateInThePast();
if (designatedStartDate > 0 && remainingPeriods <= 1) {
    // Business rule: do not allow credits with designated start date to have only 1 period.
    revert Errors.PayPeriodsTooLowForCreditsWithDesignatedStartDate();
}
```

Since the conditional statements used here have a common condition of `designatedStartDate > 0`, one can refactor this condition into an outer `if` block.

Recommendation: Since these `3` `if` blocks are of the form:

```
if (designatedStartDate > 0 && /*...*/) { /*...*/ }
```

It might be cleaner to take than conditional statement to an outer `if` block:

```
if (designatedStartDate > 0) {
    // <IF_BLOCK_1>
    // <IF_BLOCK_2>
    // <IF_BLOCK_3>
}
```

Huma: Fixed in PR 394.

Spearbit: Fixed.

5.5.23 Avoid using inequalities when comparing an enum value to a set of enum values

Severity: Informational

Context: CreditManager.sol#L226

Description: In `_approveCredit` we have:

```
if (cr.state > CreditState.Approved) revert Errors.CreditNotInStateForUpdate();
```

This approach of checking whether a state which an enum value is in a specific set is prone to future bugs in case enum elements are rearranged.

Recommendation: It would be best to be explicit with the allowed states and avoid using inequality when comparing enum values. This would avoid potential bugs in the future if the enum elements are rearranged.

Huma: Fixed in PR 392.

Spearbit: The fix does not include the `Paused` state since it is proposed to be removed. Fixed.

5.5.24 `_computeYieldNextDue` can be simplified

Severity: Informational

Context: CreditDueManager.sol#L421-L430

Description: In `_computeYieldNextDue` we are taking a maximum of a piece-wise linear function. And this case the maximum can be applied to the inputs to compute the maximum output:

$$\max\left(\left\lfloor\frac{xa}{b}\right\rfloor,\left\lfloor\frac{ya}{b}\right\rfloor\right)=\left\lfloor\max\left(\frac{xa}{b},\frac{ya}{b}\right)\right\rfloor=\left\lfloor\frac{\max(x,y)a}{b}\right\rfloor$$

Recommendation: Here is the more simplified version:

```
function _computeYieldNextDue(
    uint256 yieldInBps,
    uint256 principal,
    uint256 committedAmount,
    uint256 daysPassed
) internal pure returns (uint256 maxYieldDue) {
    // You can define a new intermediary variable below if desired
    maxYieldDue = principal > committedAmount ? principal : committedAmount;
    maxYieldDue = _computeYieldDue(maxYieldDue, yieldInBps, daysPassed);
}
```

Huma: Fixed in PR 397.

Spearbit: Fixed.

5.5.25 `updatePoolStatus` does not validate the status transition

Severity: Informational

Context: PoolFactory.sol#L545-L558, PoolFactory.sol#L35-L39, PoolFactory.sol#L620

Description: In `updatePoolStatus` the factory admin can update a pool's status to any of the below states except the `Created` status (which is the default `0` value and also used in the `deployPool` endpoint).

```
enum PoolStatus {
    Created, // the pool is created but not initialized yet
    Initialized, // the pool is initialized and ready for use
    Closed // the pool is closed and not in operation anymore
}
```

Recommendation: Either document that the above flow is allowed and the reasons for each allowed state transition in the above diagram or perhaps the flow of states should be like the following:

and all the other transitions should be disallowed.

Huma: We changed the `updatePoolStatus` function in PR 375. Key changes:

1. The permission changed from `_onlyFactoryAdmin` to `_onlyDeployer`.
2. Combined events `PoolClosed` and `PoolStatusUpdated` .

Here are the reasons behind this design:

1. The pool status in the factory is for deployers to track pool status internally, the actual operational status of the pool is declared in `Pool.sol`.
2. In case a deployer misconducts and sets a pool to a wrong status, the deployer needs to able to fix it, otherwise, the deployer can only set the status to be closed and abandon it.

3. Except for misconduct, the deployer would not want to mess up the pool statuses in the factory as it wastes gas.
4. Even if a compromised deployer comes to set wrong pool statuses, we will detect it by monitoring, and then remove the compromised deployer and fix pool statuses.

Spearbit: Acknowledged.

5.5.26 Use `_addProxy` directly and remove all the `_add<CONTRACT>(...)` internal functions

Severity: Informational

Context: PoolFactory.sol#L643-L724, PoolFactory.sol#L637-L641

Description/Recommendation: Use `_addProxy` directly and remove all the `_add<CONTRACT>(...)` internal functions for a cleaner codebase.

Huma: Fixed in PR 420.

Spearbit: Fixed.

5.5.27 `distributeLoss` endpoint can be restricted to only the `CreditManager`

Severity: Informational

Context: Pool.sol#L169-L172

Description: The current `Credit` implementations do not call into `distributeLoss`.

Recommendation: `distributeLoss` endpoint can be restricted to only the `CreditManager`.

Huma: Fixed in PR 412.

Spearbit: Fixed.

5.5.28 Document the calendar and payment/interest calculations for both lenders and borrowers

Severity: Informational

Context: FixedSeniorYieldTranchesPolicy.sol#L89-L94

Description: The yield tracker in `FixedSeniorYieldTranchePolicy` follows a `365` calendar days for the liquidity lender side whereas on the borrow side most calculations follow the `30/360` calendar pattern.

Recommendation: It would be best to document above or change the scheduling on both sides so that they would follow the same pattern.

Huma: We are using 365 here since the fixed senior yield policy is computed to the second-level, not the day-level as seen for borrower yield and late fee. But it's indeed a bit confusing that lender yield are calculated following a different schedule than the borrower yield calculation. So we will update here to compute to the day-level as well, and use 360 days-per-year everywhere.

We updated `FixedSeniorYieldTranchesPolicy` to use day-boundary calculations and got rid of the constant for 365 days in PR 410.

Spearbit: Fixed, although the new implementation uses coarser day-based calculation using a future timestamp.

5.5.29 `depositRecord.principal` calculation can be simplified

Severity: Informational

Context: TrancheVault.sol#L342-L346, TrancheVault.sol#L339, TrancheVault.sol#L308-L311

Description/Recommendation: Let:

parameter	description
p^u	`depositRecord.principal` on line 339
s	`shares` on line 339
s^u	`sharesBalance` on line 339

We know that $s \leq s^u$ so:

$$p^u - \left\lfloor \frac{p^u \cdot s}{s^u} \right\rfloor \geq p^u - \left\lfloor \frac{p^u \cdot s^u}{s^u} \right\rfloor = 0$$

so the update for p^u can be simplified to:

```
depositRecord.principal -= uint96(principalRequested)
```

Huma: Good catch. Simplified in PR 427. We might have written it this way because of rounding issues in a previous implementation of this function. We don't have rounding issues anymore in this function, but forgot to simplify the calculation accordingly.

Spearbit: Fixed.

5.5.30 `totalSharesProcessed` is used in `unprocessedAmount` calculation although it should be equal to `0`

Severity: Informational

Context: EpochManager.sol#L120-L123

Description/Recommendation: When we end up calculating the`unprocessedAmount`:

```
unprocessedAmount =
    (((seniorSummary.totalSharesRequested - seniorSummary.totalSharesProcessed) *
        seniorPrice) +
        ((juniorSummary.totalSharesRequested - juniorSummary.totalSharesProcessed) *
            juniorPrice)) /
    DEFAULT_DECIMALS_FACTOR;
```

it seems that both `seniorSummary.totalSharesProcessed` and `juniorSummary.totalSharesProcessed` equal to zero. They only get populated at the end of the `closeEpoch` flow, the epoch increments and for the new epoch `totalSharesProcessed` always starts from `0`.

Huma: We meant to calculate the unprocessed amount after processing the epoch, not before, so `totalSharesProcessed` would not be 0. This is a bug we'll fix. The impact is small though since `unprocessedAmount` is only being emitted as a parameter in the event, not used in any subsequent calculations.

The issue has been fixed as part of the "close pool" feature, in PR 418.

Spearbit: Fixed.

5.5.31 Make sure `poolConfig.onlyProtocolAndPoolOn()` is applied to all the required endpoints

Severity: Informational

Context: FirstLossCover.sol#L231, TrancheVault.sol#L421

Description/Recommendation: Make sure `poolConfig.onlyProtocolAndPoolOn()` is applied to all the required endpoints. Possibly these:

- `payoutYield`
- `processYieldForLenders`

Huma: Fixed in commit PR 424.

Spearbit: Fixed.

5.5.32 Incorrect comments

Severity: Informational

Context: Credit.sol#L324-L326, BaseTranchesPolicy.sol#L84

Description: The code contains inaccuracies and typos in comments, this affects code comprehension.

Recommendation: It's recommended to fix all comments mentioned in context would help in making the code clearer and more consistent. The corresponding fixes are:

- Credit.sol#L324-L326:

```
  // If the payment is not enough to cover the total amount past due, then
- // apply the payment to the yield past due, followed by principal past due,
+ // apply the payment to the yield past due, followed by  late fees,
- // then lastly late fees.
+ // then lastly principal past due.
```

- BaseTranchesPolicy.sol#L84:

```
- * @notice Internal function that calculates the profit distribution between the junior trnache and
+ * @notice Internal function that calculates the profit distribution between the junior tranche and
```

Huma: Fixed in PR 423. Also fixed one typo in the name of a variable and a bunch of inconsistencies in the comments.

Spearbit: Fixed.

5.5.33 Use constants instead of inline numbers

Severity: Informational

Context: PoolFactory.sol#L370-L404, PoolFactory.sol#L409-L423

Description: When using numbers, it should be made clear what the number represents by storing it as a constant variable. Use named constants instead of inline numbers for clearer, more understandable code.

Recommendation: It is recommended to replace the inline numbers with corresponding named constants.

```
  IFirstLossCoverLike(borrowerFirstLossCover).initialize(
    "Borrower First Loss Cover",
    "BFLC",
    poolConfig
  );
  _setFirstLossCover(
    poolConfigAddress,
    borrowerFirstLossCover,
-   0,
```

```
+   BORROWER_LOSS_COVER_INDEX,
    FirstLossCoverConfig(0, 0, 0, 0, 0)
  );
  address insuranceFirstLossCover = _addFirstLossCover();

  IFirstLossCoverLike(insuranceFirstLossCover).initialize(
    "Insurance First Loss Cover",
    "IFLC",
    poolConfig
  );
  _setFirstLossCover(
    poolConfigAddress,
    insuranceFirstLossCover,
-   1,
+   INSURANCE_LOSS_COVER_INDEX,
    FirstLossCoverConfig(0, 0, 0, 0, 0)
  );
  address adminFirstLossCover = _addFirstLossCover();
  IFirstLossCoverLike(adminFirstLossCover).initialize(
    "Admin First Loss Cover",
    "AFLC",
    poolConfig
  );
  _setFirstLossCover(
    poolConfigAddress,
    adminFirstLossCover,
-   2,
+   ADMIN_LOSS_COVER_INDEX,
    FirstLossCoverConfig(0, 0, 0, 0, 0)
  );
```

```
  if (i == 8) {
    IVaultLike(poolAddresses[i]).initialize(
      "Senior Tranche Vault",
      "STV",
      poolConfig,
-     0
+     SENIOR_TRANCHE
    );
  } else if (i == 9) {
    IVaultLike(poolAddresses[i]).initialize(
      "Junior Tranche Vault",
      "JTV",
      poolConfig,
-     1
+     JUNIOR_TRANCHE
    );
  } else {
```

Huma: Fixed in PR 425.

Spearbit: Fixed.

5.5.34 Remove redundant return

Severity: Informational

Context: Calendar.sol#L171, BaseTranchesPolicy.sol#L43, BaseTranchesPolicy.sol#L126, CreditDueManager.sol#L23, CreditDueManager.sol#L244, CreditDueManager.sol#L343, CreditDueManager.sol#L390, CreditDueManager.sol#L440, CreditDueManager.sol#L463, CreditDueManager.sol#L476, CreditLine.sol#L40, CreditLine.sol#L55, CreditManager.sol#L569, FixedSeniorYieldTranchePolicy.sol#L76, RiskAdjustedTranchesPolicy.sol#L36, Pool.sol#L435, PoolFeeManager.sol#L369

Description: Adding a return statement when the function defines a named return variable is redundant and can be removed.

Recommendation: It is recommended to remove the redundant returns.

Huma: Calendar functions fixed in PR 404 and other places fixed in PR 429.

Spearbit: Fixed.

5.5.35 Avoid Duplicated code

Severity: Informational

Context: CreditLine.sol#L72, ReceivableBackedCreditLine.sol#L255

Description: `ReceivableBackedCreditLine` and `CreditLine` both have `getCreditHash` function.

Recommendation: The function can be moved to a common contract to avoid duplicated code.

Huma: Given that the amount of duplicated code is small and would require structural changes to avoid duplicates, we've decided to keep things as-is for now.

Spearbit: Acknowledged.

5.5.36 Remove unused `events`, `enums`, `constants` and `errors`

Severity: Informational

Context: HumaConfig.sol#L108, Credit.sol#L54-L68, Pool.sol#L73-L82, Errors.sol#L105-L119, SharedDefs.sol#L12, ReceivableFactoringCredit.sol#L20, CreditStructs.sol#L103-L107, CreditStructs.sol#L134-L139,

Description: Some `events`, `enums`, `constants`, and `errors` have been declared but are never used in the contract's functions, consider removing or using them.

- **Events:** `HumaConfig.ProtocolDefaultGracePeriodChanged`, `Credit.CreditInitiated`, `Credit.CreditLineChanged`, `Pool.PoolAssetsRefreshed`.
- **Enums:** `CreditStructs.PaymentStatus`, `CreditStructs.CreditClosureReason`.
- **Constants:** `SharedDefs.MAX_PERIODS`, `ReceivableFactoringCredit.PAYER_ROLE`
- **Errors:** `DurationTooLong`, `InvalidFlowRate`, `OnlySuperfluid`, `BorrowerMismatch`, `FlowKeyMismatch`, `FlowIsNotTerminated`, `InvalidSuperfluidCallback`, `InvalidSuperfluidAction`, `NotTradableStreamOwner`, `TradableStreamNotExisting`, `TradableStreamNotMatured`, `InsufficientAvailableFlowRate`, `AuthorizationExpired`, `InvalidAuthorization`, `NewReceiverSameToOrigin`.

Recommendation: It's recommended to either remove or use them.

Huma: Unused credit enums removed in PR 411, other unused values removed in PR 422.

Spearbit: Fixed.

5.5.37 Floating pragma

Severity: Informational

Context: Applicable to all contracts

Description: The project contains many instances of floating pragma. To maintain consistency and avoid potential bugs, contracts should be deployed using the same compiler version and configuration as used during thorough testing. Floating pragma can lead to unintended deployments with outdated compiler versions, introducing bugs, or with versions incompatible with certain EVM chains.

Recommendation: It is recommended to lock the pragma to a specific compiler version that assures compatibility across all EVM chains, such as version `0.8.19`.

Huma: Fixed in PR 452.

Spearbit: Fixed.

5.5.38 `Counters.sol` is deprecated in most recent OZ version

Severity: Informational

Context: EvaluationAgentNFT.sol#L5

Description: The recent version of OZ library 5.0.0, removed `Counters` library. The code will break in case updating the OZ version.

Recommendation: Consider removing `Counters` library and using increment and decrement:

```
  function mintNFT(address recipient) external returns (uint256) {
-     _tokenIds.increment();
+     _tokenIds += 1;

-     uint256 newItemId = _tokenIds.current();
-     _mint(recipient, newItemId);
+     _mint(recipient, _tokenIds);

-     emit NFTGenerated(newItemId, recipient);
-     return newItemId;
+     emit NFTGenerated(_tokenIds, recipient);
+     return _tokenIds;
  }
```

Huma: Thanks for pointing it out. Reading through the discussion about the motivation behind removing it here, it looks like it was removed because it's not offering much value beyond what's natively supported by the latest version of the Solidity compiler, not because of security issues. So we consider this to be a relatively low priority and will take it on if we have enough time.

Since Counters has not been deprecated due to security concerns, its continued use should not pose any immediate security risks. Even if we upgrade the OZ version in the future when we develop the next release of the protocol, there will be no effect on contracts that have already been deployed

Spearbit: We agree not to upgrade it. Acknowledged.

6 Additional Comments

The latest commit on the `spearbit-audit` branch underwent review by all security researchers, with all identified issues addressed or acknowledged. Additionally, Kankodu reviewed the following PRs:

- PR 473
- PR 474
- PR 478

where new changes were made after the review period was over.

The Huma team indicated that during the first round of review (including the PRs that were reviewed separately), all changes were merged into `spearbit-audit-updated`, and then merged into `develop`. The commit hash for merging `spearbit-audit-updated` into develop was 50261624b15c4dd2c839fc6d1bd95db4a96883a8.

Afterwards, Kankodu reviewed every change up to commit 46791250275e0197a9624edb90dc5e61f4be76ee, which was merged into `develop`.

Then, the Huma team created a new PR to tag the commit as `v2.0.1`, and merged it into develop (58b9f3995ba208f0e4b9f72c232977a53786ea7d) and main (0f76854d79a7df1b34f749f98a14835bccdc1269) respectively.

The Huma team will be launching all the pools using the last commit hash on `main`, i.e. 0f76854d79a7df1b34f749f98a14835bccdc1269.

For more details, please see the commit history of `main`.