yAudit

March 2026

Prepared for
Alchemix - V3

Audited by
Panda
Watermelon

Alchemix - V3

Smart Contract Security Assessment

Contents

1 Review Summary

1.1 Protocol Overview

Alchemix V3 is a self-repaying DeFi lending protocol built around yield-bearing collateral and synthetic asset issuance.

1.2 Audit Scope

This audit covers 39 smart contracts totaling approximately 5,000 lines of code across 22 days of review.

```
src/
  ├── adapters
  │   ├── AbstractFeeVault.sol
  │   └── EulerUSDCAdapter.sol
  ├── AlchemistAllocator.sol
  ├── AlchemistCurator.sol
  ├── AlchemistETHVault.sol
  ├── AlchemistGate.sol
  ├── AlchemistStrategyClassifier.sol
  ├── AlchemistTokenVault.sol
  ├── AlchemistV3.sol
  ├── AlchemistV3Position.sol
  ├── AlchemistV3PositionRenderer.sol
  ├── base
  │   ├── ErrorMessages.sol
  │   ├── Errors.sol
  │   └── TransmuterErrors.sol
  ├── libraries
  │   ├── FixedPointMath.sol
  │   ├── NFTMetadataGenerator.sol
  │   ├── SafeCast.sol
  │   ├── SafeERC20.sol
  │   ├── Sets.sol
  │   ├── StakingGraph.sol
  │   └── TokenUtils.sol
  ├── MYTStrategy.sol
  ├── PerpetualGauge.sol
  ├── strategies
  │   ├── arbitrum
  │   │   ├── AaveV3ARBUSDCStrategy.sol
  │   │   ├── AaveV3ARBWETHStrategy.sol
  │   │   ├── EulerARBUSDCStrategy.sol
  │   │   ├── EulerARBWETHStrategy.sol
  │   │   └── FluidARBUSDCStrategy.sol
  │   ├── mainnet
  │   │   ├── EulerUSDCStrategy.sol
  │   │   ├── EulerWETHStrategy.sol
```

```
│   │   ├── MorphoYearnOGWETH.sol
│   │   ├── PeapodsETHStrategy.sol
│   │   ├── PeapodsUSDCStrategy.sol
│   │   ├── TokeAutoETHStrategy.sol
│   │   ├── TokeAutoUSDStrategy.sol
│   │   └── WStethStrategy.sol
│   └── optimism
│       ├── AaveV3OPUSDCStrategy.sol
│       ├── MoonwellUSDCStrategy.sol
│       └── MoonwellWETHStrategy.sol
├── Transmuter.sol
└── utils
    ├── PermissionedProxy.sol
    ├── Whitelist.sol
    └── ZeroXSwapVerifier.sol
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	5
Low	10
Informational	14

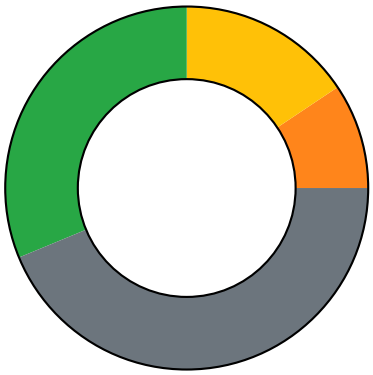

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

Alchemix V3 presents a thoughtful and feature-rich design with a responsive team that addressed the substantial majority of identified issues during the engagement. While the protocol's complexity and strategy surface area introduced several implementation risks, no critical findings were identified, and the final security posture improved meaningfully through prompt remediation, iteration and refactoring.

2 Audit Overview

2.1 Project Information

Protocol Name: Alchemix - V3
Repository: https://github.com/alchemix-finance/v3/
Commit Hash: 45e4b0852485970fc2b7cae24735c6463923cf86
Commit URL:
https://github.com/alchemix-finance/v3/blob/45e4b0852485970fc2b7cae24735c6463923cf86

2.2 Audit Team

Panda, Watermelon

2.3 Audit Timeline

The audit was conducted from February 16 to March 18, 2026.

2.4 Audit Resources

Code repositories and documentation Additional resources: whitepaper, previous audits, etc.

Category	Mark	Description
Access Control	Average	Access control design is serviceable but not robust. The multicall-based PermissionedProxy bypass showed that wrapper-level restrictions could be sidestepped.
Mathematics	Average	Core accounting and collateral math are non-trivial and several fee, valuation, and collateralization edge cases were missed. Liquidation-fee logic, bad-debt redemption accounting, and multiple strategy valuation paths all required corrective changes.
Complexity	Low	The codebase is highly complex, with a large monolithic core contract, many strategy-specific branches, and repeated adapter implementations. That complexity materially contributed to edge-case risk and maintainability issues.
Libraries	Good	The project generally relies on standard components and established integrations.
Decentralization	Good	The protocol retains meaningful privileged control over risk parameters, strategy permissions, and liquidation-related configuration. No critical governance flaw was identified, but safety still depends heavily on trusted operators and sound admin decisions.
Code Stability	Average	The codebase was actively iterated during the review and most findings were fixed promptly. That responsiveness is positive, but the number and spread of fixes indicate the implementation was still maturing across core flows and strategy integrations.
Documentation	Good	No material documentation weakness was identified during the review. The available documentation and inline context were sufficient for assessment, and the main findings were concentrated in implementation correctness rather than documentation quality.
Monitoring	Good	No material monitoring weakness was identified during the review. The codebase emits events across core protocol flows, and event emission was not a meaningful source of risk relative to the implementation issues identified.
Testing and verification	Average	Testing appears reasonably developed, with PoCs and regression tests added for confirmed issues, but pre-existing coverage did not catch a number of important liquidation, accounting, and integration edge cases. Coverage quality is acceptable, not exceptional.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

2.6.1 Repayment-fee short-circuit enables double payout and forced liquidation

Technical Description

When `_liquidate()` processes an unhealthy account that has earmarked debt, it first calls `_forceRepay()` to repay the earmarked portion using account collateral. If the account becomes healthy after repayment, the function enters the short-circuit branch: it computes a repayment fee, deducts it from the account's collateral, transfers it to the liquidator, and returns without performing a full liquidation.
The problem is that `_liquidate()` does not re-check account health after deducting the repayment fee from collateral. The fee is computed by `_calculateRepaymentFee()` based on surplus above 100% debt backing (`collateralBalance - debtInYield`), not surplus above the protocol's `collateralizationLowerBound`. For accounts that are barely healthy after `_forceRepay()`, the fee deduction can reduce collateral below the health threshold while the function has already committed to the short-circuit return path.
A liquidator can then immediately call `liquidate()` a second time on the same account. With the earmarked debt already cleared, the second call bypasses the `_forceRepay` branch entirely and proceeds directly to `_doLiquidation()`.
The result is that the liquidator may collect two separate fees across two calls: a repayment fee (in yield tokens, from the account's collateral) on the first call, and an outsourced liquidation fee (in underlying tokens, from the fee vault) on the second.

Proof of Concept

Create `src/test/poc/PoC_liquidation_fee_double_dip.t.sol`, insert the content shared in the following Github Gist and run with
`forge t --mt test_PoC_LiquidationFeeDoubleDip -vvv`
gist

Impact

High. An account that should have been restored to health by the earmarked debt repayment is instead fully liquidated. The account owner loses their entire position when only a small earmark repayment was needed. The liquidator extracts two fees: the repayment fee from the victim's collateral and an outsourced liquidation fee from the protocol's fee vault.

Recommendation

After deducting the repayment fee in `_liquidate()`, re-check account health. If the fee broke health, either clamp the fee to preserve the health invariant or fall through to the full liquidation path instead of returning early.

Alternatively, compute the repayment fee relative to the collateralization threshold rather than 100% debt backing, which would reduce the fee amount and make it less likely to breach health.

Developer Response

Fixed in commit : cbe70ed by modifying repayment fee calculation to be based on the collateralizaiton threshold rather than 100% debt backing.

2.6.2 Bad-debt haircut bypass via temporary collateral deposit

During bad debt, a user can temporarily increase reported backing with a zero-debt MYT deposit, claim redemption with a smaller haircut, then withdraw that same MYT. The system returns to the same insolvency level, but the attacker keeps excess payout.
Technical Details

- `claimRedemption()` scales payout using `badDebtRatio = totalSyntheticsIssued / backing`.
- backing includes `getTotalLockedUnderlyingValue()`.
- `getTotalLockedUnderlyingValue()` is capped by current held shares (_mytSharesDeposited), so temporary deposits increase backing immediately.
- `deposit()` allows adding collateral without minting debt.
- `withdraw()` only checks position-level collateralization and allows full withdrawal from zero-debt positions.
- As a result, an attacker can:

1. Deposit MYT into a zero-debt position.
2. Call claimRedemption() and receive a reduced haircut.
3. Withdraw the temporary MYT deposit.

Impact

High. - Unfair loss distribution during undercollateralization. - Attackers can extract more MYT than their fair haircut-adjusted share. - Remaining synthetic holders absorb larger losses. - Feasible with flash loaned liquidity.

Recommendation

- Enforce a global withdrawal cap in withdraw():

1. Compute positionFree.
2. Compute globalFree = max(_mytSharesDeposited - _requiredLockedShares(), 0).
3. Require amount <= min(positionFree, globalFree).

Developer Response

The issue we are having is that blocking users from withdrawing who have no debt in the system is not really an option. If they are personally not contributing to bad debt it wouldnt really be fair to do this.

We are leaning towards the ”Disallow users from depositing or minting during bad debt. This leaves us with one edge case where a user who is already deposited into the alchemist with no debt that also has a transmuter position can claim and then withdraw even if that puts the protocol in bad debt.” option.
Fixed in commit : bb5e30e.

2.6.3 `PermissionedProxy` selector filter bypass via `VaultV2.multicall` enables blocked function execution

`PermissionedProxy.proxy` only validates the outer calldata selector. Because `permissionedCalls` is enforced as a denylist (`require(!permissionedCalls[selector], "PD")`), an operator can call `vault.multicall(...)` (not blocked) and tunnel blocked calls like `vault.allocate(...)` inside it. This bypasses wrapper-level safeguards such as `_validateCaps`.

Technical Details

`PermissionedProxy.proxy` checks only `bytes4(data)` of the top-level payload (`src/utils/PermissionedProxy.sol:63` - `src/utils/PermissionedProxy.sol:69`). In `AlchemistAllocator`, raw `allocate/deallocate` selectors are blocked in `permissionedCalls` (`src/AlchemistAllocator.sol:27` - `src/AlchemistAllocator.sol:29`) to force use of wrapper methods that run `_validateCaps` (`src/AlchemistAllocator.sol:41` - `src/AlchemistAllocator.sol:48`, `src/AlchemistAllocator.sol:122` - `src/AlchemistAllocator.sol:149`).
By sending `vault.multicall([abi.encodeCall(vault.allocate, ...)])` through `proxy`:

1. Outer selector is `multicall`, which is not blocked.
2. `VaultV2.multicall` executes inner payloads via `delegatecall` (`lib/vault-v2/src/VaultV2.sol:282` - `lib/vault-v2/src/VaultV2.sol:285`).
3. `msg.sender` remains the allocator wrapper during inner execution, so `VaultV2.allocate` authorization passes (`lib/vault-v2/src/VaultV2.sol:566` - `lib/vault-v2/src/VaultV2.sol:568`).
4. `_validateCaps` in `AlchemistAllocator` is never reached.

Impact

High. Operators can execute blocked vault functions by wrapping them in `multicall`, bypassing intended access/risk controls. For allocator flow, this allows bypassing classifier cap enforcement and allocating above wrapper-imposed limits.

Recommendation

Block proxying `multicall` or use a whitelist approach.

Developer Response

Changed to whitelist on b273fcb

2.7 Medium Findings

2.7.1 Repayment fee is charged on 100% surplus basis instead of repaid amount

Technical Details

In the repayment-fee path of `AlchemistV3._liquidate`, fee calculation calls `_calculateRepaymentFee(accountId, repaidAmountInYield)`.
When account surplus exists, `_calculateRepaymentFee` computes:

- `surplus = account.collateralBalance - convertDebtTokensToYield(account.debt)`
- `feeInYield = surplus * repaymentFee / BPS`

So the fee is charged on collateral above 100% backing (`collateral - debtInYield`), not on the actual amount repaid in the call (`repaidAmountInYield`).
This can produce repayment fees that are much larger than the configured repayment fee rate applied to the repaid amount.
Example shape:

1. `repaidAmountInYield` is small.
2. `surplus` is large.
3. Liquidator fee becomes `repaymentFee%` of large `surplus`, not `repaymentFee%` of small repay.

Regression reference:

- `src/test/LiquidationRepaymentFeeRegression.t.sol`
- `test_repayment_fee_can_exceed_fee_rate_applied_to_repaid_amount`

Impact

Medium. Liquidators can extract significantly more fee than expected under a repay-proportional model.

Recommendation

- Base repayment fee on `repaidAmountInYield`.

Developer Response

Fixed in 73a165a.

2.7.2 `AlchemistAllocator._validateCaps` risk caps check per-transaction amount instead of cumulative allocation

Technical Description

`AlchemistAllocator._validateCaps` enforces four cap layers on allocations: the vault's `absoluteCap`, `relativeCap`, a `globalRiskCap`, and a `localRiskCap`. The vault enforces the first two cumulatively in `VaultV2.allocateInternal` by tracking `caps[id].allocation` after each operation. The risk caps, however, are only enforced by the Allocator.
The Allocator compares the per-transaction `amount` against the cap ceiling:

```
require(amount <= limit, EffectiveCap(amount, limit));
```

It does not account for existing allocation. If `localRiskCap = 1_000 ether` and the vault's `absoluteCap` is 10*000 ether, an operator can call `allocate(adapter, 999 ether)` ten times. Each call passes because `999 <= 1_000`, but the strategy accumulates 9*990 ether, far exceeding the intended risk cap.

Impact

Medium. Operators can exceed per-strategy caps through repeated allocations when vault caps are set wider than risk caps.

Recommendation

Read the current allocation via `vault.allocation(id)` and check the cumulative value:

```
uint256 currentAllocation = vault.allocation(id);
require(currentAllocation + amount <= limit, EffectiveCap(amount, limit));
```

Developer Response

fixed in
https://github.com/alchemix-finance/v3/commit/6b941f448d2efdc7d9263c7902aa83bc207eec3e

2.7.3 `AlchemistAllocator` does not enforce `globalCap`

Technical Details

`AlchemistStrategyClassifier` documents `globalCap` as the maximum allocation for **all strategies of a risk class combined**:

- `AlchemistStrategyClassifier.sol:16` says `globalCap` is the maximum allocation for "ALL strategies of this risk type combined".

However, `AlchemistAllocator._validateCaps()` does not enforce that aggregate property. Instead, it fetches the risk level and the corresponding `globalRiskCap`, then applies it only as a limit on the **current call amount**:

```
uint8 riskLevel = strategyClassifier.getStrategyRiskLevel(strategyId);
uint256 globalRiskCap = strategyClassifier.getGlobalCap(riskLevel);

uint256 limit = absoluteCap < absoluteValueOfRelativeCap ? absoluteCap :
absoluteValueOfRelativeCap;
limit = limit < globalRiskCap ? limit : globalRiskCap;

require(amount <= limit, EffectiveCap(amount, limit));
```

Critically, this logic never:

1. reads the current allocation of the target strategy,
2. sums allocations of other strategies in the same risk bucket, or
3. checks whether `existingRiskBucketExposure + amount` exceeds `globalCap`.

As a result, the same `globalCap` can be consumed repeatedly across multiple strategies that share the same risk class.

Impact

Medium. The global cap for a risk class is not enforced.

Recommendation

Enforce `globalCap` against all the strategies in a risk class.

Developer Response

fixed in 71d0f746 with corresponding test.

2.7.4 Impossible WETH to wstETH swap

Technical Details

In `WstethMainnetStrategy` swap allocation:

- `WStethStrategy.sol:67` calls:
  `dexSwap(address(wsteth), address(weth), amount, amount, callData);`
- `WStethStrategy.sol:68` then enforces: `require(wethOut == amount, "IA");`

This logic is invalid for two independent reasons:

1. Token economics:

- `wstETH` is a wrapped share token whose value per unit increases over time versus stETH.
- Therefore, swapping `1 WETH -> wstETH` should normally return less than `1 wstETH`.

2. Wrong invariant on swap return value:

- `dexSwap` returns actual output (`amountReceived`), not input amount.
- Enforcing `out == amount` is therefore incorrect for market swaps in general, even if route/token changes (e.g., swapping to stETH), because output naturally differs due to price, fees, and rounding.

As a result, this `ActionType.swap` allocation path reverts in practice.

Impact

Medium. `ActionType.swap` allocation for this strategy is non-functional due to persistent revert conditions.

Recommendation

- Replace `minAmountOut = amount` with a realistic quote-based minimum (e.g., quote `minBuyAmount`).
- Remove strict equality `out == amount`; validate `out >= minAmountOut`.

Developer Response

Fixed in several commits b273d.

2.7.5 WstETH strategy misprices stETH and omits idle WETH

Technical Details

WstethMainnetStrategy reports value in stETH terms, not executable WETH terms:

- _totalValue() uses wsteth.getStETHByWstETH
- This assumes 1 stETH == 1 WETH economically, but real exits require stETH -> WETH swap with fee/slippage.

So even at peg, realizable WETH is lower than stETH notional due to swap costs. Additionally, deallocation can leave surplus WETH idle:

- Swap may return wethReceived > amount (WStethStrategy.sol:94.
- Only `amount` is approved to the vault.
- `_totalValue()` does not include idle WETH, so held assets are underreported.

Impact

Medium. - Systematic accounting mismatch between reported and realizable value. - Can overstate strategy value (stETH notional vs net WETH after swap costs). - Can understate value when idle WETH surplus accumulates and is ignored.

Recommendation

- Value in vault asset terms (WETH), not raw stETH notional.
- Include idle WETH balance in _totalValue().

Developer Response

Fixed in 1b676

2.8 Low Findings

2.8.1 MoonwellStrategy uses stale `exchangeRateStored()` for redemption and valuation

Technical Details

`MoonwellStrategy.sol:65` and `MoonwellStrategy.sol:72` use `mToken.exchangeRateStored()` to value newly minted shares and to compute the number of shares that must be redeemed. `MoonwellStrategy.sol:88` and `MoonwellStrategy.sol:95` also rely on the same stale rate for reporting total strategy value. On Moonwell, `exchangeRateStored()` does not accrue interest, while `exchangeRateCurrent()` updates the rate first. If interest has accrued since the last market update, the strategy will operate on an outdated exchange rate. In that state, `deallocate()` can redeem too many shares, leave unexpected idle underlying stranded in the strategy, and cause `_totalValue()` to underreport assets. A follow-up withdrawal can then revert because the strategy believes it still holds enough mTokens when it does not.

Impact

Low. Using `exchangeRateStored()` can cause the strategy to misprice its Moonwell position.

Recommendation

Use `exchangeRateCurrent()`. Use `redeemUnderlying` when deallocating to remove the need to call `exchangeRateCurrent`.

Developer Response

`exchangeRateCurrent()` calls `accrueInterest()` first though which makes it state-mutable, breaking the full call signature chain assuming the query to be `view`.
We addressed the issue in commit 040016f by calling `accrueInterest` during deallocation just before `exchangeRateStored()` - which is exactly what `exchangeRateCurrent()` does internally.

2.8.2 ERC4626 strategies use `convertToAssets` instead of `previewRedeem` in `_totalValue`

Technical Description

All strategies integrating with an ERC4626-compliant vault use `convertToAssets` within their `_totalValue` implementation to calculate the amount of underlying assets held by the strategy. The affected strategies are:

- `EulerUSDCStrategy`
- `EulerWETHStrategy`
- `EulerARBUSDCStrategy`
- `EulerARBWETHStrategy`
- `PeapodsUSDCStrategy`
- `PeapodsETHStrategy`
- `MorphoYearnOGWETHStrategy`

Notice that the following strategies are unaffected for various reasons:

- `FluidARBUSDCStrategy`: the current implementation's `previewRedeem` wraps `converToAssets`, so no meaningful difference is present between the two methods.
- `TokeAutoETHStrategy` and `TokeAutoUSDStrategy`: the current implementations do not expose the `previewRedeem` method.

Per the ERC4626 specification, `convertToAssets` provides a mathematical conversion between shares and assets without accounting for protocol-specific fees or withdrawal conditions. In contrast, `previewRedeem` is designed to return the actual amount of assets that would be received upon redeeming a given number of shares, inclusive of any fees charged by the underlying vault.
For vaults that charge withdrawal or redemption fees, these two functions can return different values. Using `convertToAssets` causes `_totalValue` to overestimate the strategy's redeemable value by the fee amount. Since `_totalValue` feeds into the MYT's `convertToAssets`, which in turn feeds into the Alchemist's collateral valuation and health calculations, this overestimation propagates through the system. The result is that positions appear slightly healthier than they are in practice, as the reported collateral value includes assets that would be lost to fees upon actual redemption.

Impact

Low. The magnitude of drift is bounded by the fee percentage charged by the integrated vault, which is typically small.

Recommendation

Replace `convertToAssets` with `previewRedeem` in `_totalValue` across all affected strategies. For example:

```
    function _totalValue() internal view override returns (uint256) {
-       return vault.convertToAssets(vault.balanceOf(address(this)));
+       return vault.previewRedeem(vault.balanceOf(address(this)));
    }
```

Developer Response

Fixed in commit: e1a2283.

2.8.3 Liquidator fees silently reduced or zeroed when fee vault is unset or underfunded

Technical Description

The `_payWithFeeVault` method is responsible for paying liquidators their fee in underlying tokens whenever the fee cannot be sourced from the liquidated account's collateral.
The method silently reduces the liquidator's reward in two scenarios:

1. Fee vault not set (`alchemistFeeVault == address(0)`): the method returns `0`, discarding the entire calculated fee.
2. Fee vault underfunded (`vaultBalance < amountInUnderlying`): the method clamps the payout to the available balance, paying the liquidator less than the protocol calculated they are owed.

```
function _payWithFeeVault(uint256 amountInUnderlying) internal returns (uint256) {
    if (alchemistFeeVault == address(0)) return 0;              // @audit full fee lost
    uint256 vaultBalance = IFeeVault(alchemistFeeVault).totalDeposits();
    if (vaultBalance > 0) {
        uint256 adjustedAmount = amountInUnderlying > vaultBalance
            ? vaultBalance                                       // @audit partial fee
            : amountInUnderlying;
        IFeeVault(alchemistFeeVault).withdraw(msg.sender, adjustedAmount);
        return adjustedAmount;
    }
    return 0;                                                    // @audit full fee lost
}
```

In all call sites, the returned value from `_payWithFeeVault` overwrites the originally calculated fee amount. The difference between what was owed and what was paid is permanently lost with no on-chain record.
This is particularly concerning in the first 2 if-clauses in `calculateLiquidation`, where the account is deeply underwater (`debt >= collateral` or global under-collateralization). In these scenarios the entire liquidator fee is outsourced to the fee vault since there is no surplus collateral to fund it. If the fee vault cannot cover the fee, the liquidator performs the liquidation, bears the gas cost, and receives little to no compensation. There is no mechanism for the liquidator to claim the outstanding balance once the fee vault is replenished.

Impact

Low. Liquidators who perform critical protocol maintenance may receive significantly less compensation than the protocol's own fee math determines they are owed. Over time, this can

disincentivize liquidation activity, especially for cases where accounts are deeply under-collateralized and the fee vault is the sole source of liquidator compensation.

Recommendation

Implement a claims-based accounting system that tracks outstanding liquidator fees when the fee vault cannot fully cover the calculated amount. When the fee vault is replenished, liquidators should be able to claim their previously owed balances.
At minimum, the protocol should emit an event whenever a fee is reduced so that off-chain systems can track the shortfall.

Developer Response

Fixed in commit: 1173fba Chose the solution: "the protocol should emit an event whenever a fee is reduced so that off-chain systems can track the shortfall".

2.8.4 Unsafe collateralization configuration allows immediately liquidatable minting

Technical Details

`AlchemistV3` allows governance to set `collateralizationLowerBound` equal to `minimumCollateralization`.

- `setCollateralizationLowerBound(value)` can be set with `value <= minimumCollateralization`.

If governance sets:

```
collateralizationLowerBound == minimumCollateralization
```

then a position can be minted exactly at the boundary and still fail the liquidation health check immediately afterward.
As a result, under this configuration:

- A user can call `mint(getMaxBorrowable())` successfully and still be instantly liquidatable in the same state, and
- An approved spender can do the same through `mintFrom()` and get immediatly position liquidated.

The `mintFrom()` case is more concerning because a spender can receive the newly minted debt tokens while the position owner bears the liquidation risk.

Impact

Low. This issue is configuration-dependent.

Recommendation

Reject unsafe threshold combinations.
Change `setCollateralizationLowerBound()` to require:
`collateralizationLowerBound < minimumCollateralization`

Developer Response

Fixed in 10c6cb.

2.8.5 Moonwell deallocation uses stale exchange rate

Technical Details

`MoonwellUSDCStrategy` and `MoonwellWETHStrategy` size redemptions with `exchangeRateStored()` and then execute `redeem(mTokensNeeded)`.

- In `_deallocate(amount)`, they compute:
  `mTokensNeeded = ceil(amount * 1e18 / exchangeRateStored())`
- `redeem()` then executes with the market's up-to-date accrued state, which can use a higher effective exchange rate than `exchangeRateStored()`.

When that happens, redemption returns more underlying than needed for `amount`. The strategy only approves/transfers `amount` back to the vault, so the excess remains idle in the strategy.
At the same time, `_totalValue()` reports only:
`mTokenBalance * exchangeRateStored() / 1e18` and does not include idle underlying held directly by the strategy. This causes under-reporting of real assets.
Over repeated deallocations, idle underlying can accumulate while reported allocation drifts downward. Later deallocations may revert if `mTokensNeeded` (computed from requested amount, ignoring idle balance) exceeds remaining mTokens, even though total assets (`idle + mToken value`) are sufficient.

Impact

Low.

- Per-call over-redemption is typically small, but it creates accounting drift over time.
- Idle underlying remains uninvested (yield drag).
- In some flows, deallocation can revert due to sizing against mTokens only, requiring operator intervention (for example, sweeping idle funds) to restore normal operation.

Recommendation

1. In `_deallocate`, consume idle underlying first.
2. Redeem only the shortfall from Moonwell.
3. Include idle underlying in `_totalValue()`: `idleUnderlying + mTokenValue`.

Developer Response

Fixed in 1da3fd.

2.8.6 `withdraw` transfers more shares than debited from state when `collateralBalance` exceeds `_mytSharesDeposited`

Technical Description

`AlchemistV3.withdraw` performs a pre-flight collateral check before calling `_subCollateralBalance`, but the two operations read the collateral in different states. This ordering creates an accounting divergence when `account.collateralBalance` exceeds `_mytSharesDeposited`.
The guard within `AlchemistV3.withdraw` reads the raw stored balance:

```
_checkArgument(_accounts[tokenId].collateralBalance - lockedCollateral >= amount);
_subCollateralBalance(amount, tokenId);
```

`AlchemistV3._subCollateralBalance` then reconciles the local balance against the global tracker before computing how many shares to remove:

```
if (collateralBalance > _mytSharesDeposited) {
    collateralBalance = _mytSharesDeposited;          // reconcile down
    account.collateralBalance = collateralBalance;
}
uint256 amountToRemove = amountInYieldTokens > collateralBalance
    ? collateralBalance                               // clamp
    : amountInYieldTokens;
account.collateralBalance = collateralBalance - amountToRemove;
_mytSharesDeposited -= amountToRemove;
return amountToRemove;                                // ignored by caller
```

When drift exists, the reconciliation step clamps `amountToRemove` to a value smaller than the requested `amount`. However, `withdraw` ignores the return value entirely and always transfers the original `amount`:

```
TokenUtils.safeTransfer(myt, recipient, amount);
```

Three conditions compound to make this exploitable without any safeguard catching it:

1. The pre-check passes because it reads the inflated stored balance, not the reconciled one.
2. The state debit is smaller than the transfer because `amountToRemove` is clamped and its return value is discarded.

Impact

Low. A user can withdraw more vault shares than the protocol's accounting records under specific system conditions, granted his position stays adequately collateralized, in the case it has non-zero debt.

Recommendation

Reconcile the account's collateral balance before performing the pre-check, and transfer `amountToRemove` rather than the original `amount`:

```
 function withdraw(uint256 amount, address recipient, uint256 tokenId) external
 returns (uint256) {
     ...
     _earmark();
     _sync(tokenId);

+    // Reconcile stored balance against global tracker before any check or
mutation.
+    if (_accounts[tokenId].collateralBalance > _mytSharesDeposited) {
+        _accounts[tokenId].collateralBalance = _mytSharesDeposited;
+    }

     uint256 debtShares = convertDebtTokensToYield(_accounts[tokenId].debt);
     uint256 lockedCollateral = FixedPointMath.mulDivUp(debtShares,
minimumCollateralization, FIXED_POINT_SCALAR);
     _checkArgument(_accounts[tokenId].collateralBalance - lockedCollateral >=
amount);
-    _subCollateralBalance(amount, tokenId);
+    uint256 transferred = _subCollateralBalance(amount, tokenId);

     _validate(tokenId);

-    TokenUtils.safeTransfer(myt, recipient, amount);
+    TokenUtils.safeTransfer(myt, recipient, transferred);

-    emit Withdraw(amount, tokenId, recipient);
-    return amount;
+    emit Withdraw(transferred, tokenId, recipient);
+    return transferred;
 }
```

This eliminates all three compounding issues: the pre-check operates on the reconciled balance, the state debit and the transfer always agree, and any drift is surfaced as a clean revert at the check rather than a silent over-transfer.

Developer Response

Fixed in commit 1e56b75.

2.8.7 `AlchemistV3.setMinimumCollateralization` emits an event with stale data

Technical Description

`AlchemistV3.setMinimumCollateralization` allows the admin to configure the minimum collateralization ratio. Before persisting the caller-supplied `value`, the function applies two caps:

1. The value is capped at `globalMinimumCollateralization` if it exceeds it.
2. The value is capped at `liquidationTargetCollateralization` if the result of the first cap still exceeds it.

However, the event `MinimumCollateralizationUpdated` is emitted using the original, uncapped `value` argument rather than the actual value written to `minimumCollateralization`:

```
function setMinimumCollateralization(uint256 value) external onlyAdmin {
    _checkArgument(value >= FIXED_POINT_SCALAR);

    // cannot exceed global minimum
    minimumCollateralization = value > globalMinimumCollateralization ?
globalMinimumCollateralization : value;

    // cannot exceed liquidation target
    if (minimumCollateralization > liquidationTargetCollateralization) {
        minimumCollateralization = liquidationTargetCollateralization;
    }

    emit MinimumCollateralizationUpdated(value); // @audit emits input, not stored value
}
```

When either cap is triggered, off-chain infrastructure that indexes `MinimumCollateralizationUpdated` events will record a value that differs from the one actually stored in the contract.

Impact

Low. The on-chain state is set correctly. The discrepancy only affects event consumers such as subgraphs, monitoring systems, and frontends that reconstruct protocol state from logs without re-reading the contract.

Recommendation

Emit the stored `minimumCollateralization` value instead of the raw input `value`:

```
-    emit MinimumCollateralizationUpdated(value);
+    emit MinimumCollateralizationUpdated(minimumCollateralization);
```

Developer Response

Fixed in commit: bc94cf2

2.8.8 `MoonwellWETHStrategy._allocate` and `MoonwellUSDCStrategy._allocate` return total strategy position value instead of the amount allocated in the call

Technical Description

`MYTStrategy._allocate` carries an explicit NatSpec documentation:

> `uint256 amount returned should be equal to the amount parameter passed in`

Both Moonwell strategies violate this. After calling `mToken.mint(amount)` they read the strategy's entire mToken balance and convert it to underlying, not just the tokens minted in the current call.

```
function _allocate(uint256 amount) internal override returns (uint256) {
    require(TokenUtils.safeBalanceOf(address(weth), address(this)) >= amount, "...");
    TokenUtils.safeApprove(address(weth), address(mWETH), amount);
    uint256 errorCode = mWETH.mint(amount);
    if (errorCode != 0) { revert MoonwellWETHStrategyMintFailed(errorCode); }
    // reads full balance, not just newly minted tokens
    uint256 mTokenBalance = mWETH.balanceOf(address(this));
    uint256 exchangeRate = mWETH.exchangeRateStored();
    return (mTokenBalance * exchangeRate) / 1e18;
}
```

`MoonwellUSDCStrategy._allocate` is identical, using `mUSDC.balanceOf(address(this))` in place of `mWETH`.
By contrast, `AaveV3ARBUSDCStrategy._allocate` and `EulerARBWETHStrategy._allocate` correctly return `amount` directly.
The return value is stored in `amountAllocated` and emitted immediately by `MYTStrategy.allocate`:

```
amountAllocated = _allocate(assets);
// ...
emit Allocate(amountAllocated, address(this));
```

After the first deposit, every subsequent `Allocate` event for a Moonwell strategy will carry the full accumulated position value rather than the incremental amount deposited in that call.

Impact

Low.

Recommendation

Snapshot the mToken balance before the mint and compute the delta to return only the newly minted tokens converted to underlying. Apply this fix to both strategies:

```
// MoonwellWETHStrategy._allocate
  function _allocate(uint256 amount) internal override returns (uint256) {
      require(TokenUtils.safeBalanceOf(address(weth), address(this)) >= amount,
 "Strategy balance is less than amount");
      TokenUtils.safeApprove(address(weth), address(mWETH), amount);
+     uint256 mTokenBalanceBefore = mWETH.balanceOf(address(this));
      uint256 errorCode = mWETH.mint(amount);
      if (errorCode != 0) { revert MoonwellWETHStrategyMintFailed(errorCode); }
-     uint256 mTokenBalance = mWETH.balanceOf(address(this));
-     uint256 exchangeRate = mWETH.exchangeRateStored();
-     return (mTokenBalance * exchangeRate) / 1e18;
+     uint256 mTokensMinted = mWETH.balanceOf(address(this)) -
mTokenBalanceBefore;
+     uint256 exchangeRate = mWETH.exchangeRateStored();
+     return (mTokensMinted * exchangeRate) / 1e18;
  }
```

```
// MoonwellUSDCStrategy._allocate
  function _allocate(uint256 amount) internal override returns (uint256) {
      require(TokenUtils.safeBalanceOf(address(usdc), address(this)) >= amount,
 "Strategy balance is less than amount");
```

```
        TokenUtils.safeApprove(address(usdc), address(mUSDC), amount);
+       uint256 mTokenBalanceBefore = mUSDC.balanceOf(address(this));
        uint256 errorCode = mUSDC.mint(amount);
        if (errorCode != 0) { revert MoonwellUSDCStrategyMintFailed(errorCode); }
-       uint256 mTokenBalance = mUSDC.balanceOf(address(this));
-       uint256 exchangeRate = mUSDC.exchangeRateStored();
-       return (mTokenBalance * exchangeRate) / 1e18;
+       uint256 mTokensMinted = mUSDC.balanceOf(address(this)) -
mTokenBalanceBefore;
+       uint256 exchangeRate = mUSDC.exchangeRateStored();
+       return (mTokensMinted * exchangeRate) / 1e18;
    }
```

This makes both Moonwell strategies consistent with `AaveV3ARBUSDCStrategy` and `EulerARBWETHStrategy`, which return `amount` directly, and satisfies the `_allocate` NatSpec documentation.

Developer Response

Fixed in ce1507e

2.8.9 `AaveV3ARBUSDCStrategy` and `AaveV3ARBWETHStrategy` silently discard non-ARB reward tokens

Technical Description

Both `AaveV3ARBUSDCStrategy._claimRewards` and `AaveV3ARBWETHStrategy._claimRewards` accept an arbitrary `token` address, representing the collateral asset whose incentive rewards to claim, and forward it to `rewardsController.claimAllRewardsToSelf`. However, both functions hardcode their balance-delta tracking to the ARB token regardless of which reward token is actually distributed:

```
uint256 arbBefore = ARB.balanceOf(address(this));
rewardsController.claimAllRewardsToSelf(assets);
uint256 arbReceived = ARB.balanceOf(address(this)) - arbBefore;

// note: 0x912CE59144191C1204E64559FE8253a0e49E6548 (arb)
// is the only current supported reward token in the aave
// incentive controller, but this can change in the future
if (arbReceived == 0) return 0;
```

If the Aave rewards controller distributes incentives in any token other than ARB, `claimAllRewardsToSelf` will successfully transfer those tokens into the strategy contract, but `arbReceived` will remain zero. The early return is then triggered, causing the function to skip the DEX swap and the transfer back to the vault entirely.
Because `MYTStrategy.withdrawToVault` exclusively sweeps `MYT.asset()` and no generic token rescue function exists in `MYTStrategy`, the claimed non-ARB reward tokens become permanently stuck in the strategy contract. Notably, the inline comment in both affected functions already acknowledges that the set of supported reward tokens may expand in the future.

Impact

Low. No funds are at risk under the current Aave rewards configuration, as ARB is presently the only distributed incentive token. However, should the rewards controller introduce additional reward tokens, any call to `claimRewards` for those assets would silently succeed on-chain while leaving the received tokens irrecoverable.

Recommendation

Replace the hardcoded ARB balance-delta tracking in both strategies with a measurement of the token actually being claimed. Because `rewardsController.claimAllRewardsToSelf` returns the list of claimed tokens and amounts, the simplest fix is to use its return value directly. The subsequent swap should then use `rewardToken` and `rewardReceived` rather than the hardcoded `ARB` reference.
Alternatively, adding a permissioned `rescueToken` or `executeDexSwap` function to `MYTStrategy` would provide a recovery path for any future tokens that arrive in the contract through this or other unforeseen paths.

PoC
Add the following test case to `test/strategies/AaveV3ARBUSDCStrategy.t.sol` and execute it with `forge t --mt test_poc_nonArb_rewards_stuck_in_strategy`

```
function test_poc_nonArb_rewards_stuck_in_strategy() public {
    address WETH = 0x82aF49447D8a07e3bd95BD0d56f35241523fBab1; // Arbitrum WETH
    uint256 WETH_REWARD = 1e18;

    // allocate to create a realistic aave position
    bytes memory params = getVaultParams();
    uint256 amountToAllocate = 1000e6;
    deal(testConfig.vaultAsset, strategy, amountToAllocate);
    vm.prank(vault);
    IMYTStrategy(strategy).allocate(params, amountToAllocate, "", address(vault));

    // etch a MockRewardsController that distributes WETH
    MockRewardsController mockRC = new MockRewardsController(WETH, WETH_REWARD);
    vm.etch(REWARDS_CONTROLLER, address(mockRC).code);
    deal(WETH, REWARDS_CONTROLLER, WETH_REWARD);

    // snapshot balances before the claim
    uint256 strategyWethBefore = IERC20(WETH).balanceOf(strategy);
    uint256 vaultUsdcBefore = IERC20(USDC).balanceOf(vault);

    // claimRewards sees arbReceived == 0 and returns early
    vm.prank(address(1)); // strategy owner
    uint256 received = IMYTStrategy(strategy).claimRewards(AAVE_V3_USDC_ATOKEN, hex"01",
 0);

    // weth is received and held in strategy
    assertEq(received, 0, "claimRewards should have returned 0 (early exit)");
    assertEq(IERC20(USDC).balanceOf(vault), vaultUsdcBefore, "Vault USDC balance should
be unchanged");
    assertEq(IERC20(WETH).balanceOf(strategy), strategyWethBefore + WETH_REWARD, "WETH
reward should be stuck in strategy");

    // withdrawToVault only sweeps vault asset
    vm.prank(address(1)); // strategy owner
    MYTStrategy(strategy).withdrawToVault();
```

```
34      // WETH is still in strategy
35      assertEq(IERC20(WETH).balanceOf(strategy), strategyWethBefore + WETH_REWARD, "WETH
    should remain stuck after withdrawToVault");
36  }
```

Developer Response

Went with option 2 that allows the auxiliary tokens to be convert into `myt.asset()` : 9dfd50b

2.8.10 `AaveV3ARBUSDCStrategy._deallocate` and `AaveV3ARBWETHStrategy._deallocate` fail to verify token balance delta after withdrawal

Technical Description

Both Aave V3 Arbitrum strategies record a pre-withdrawal balance snapshot before calling `pool.withdraw`, but neither snapshot is used in the post-withdrawal assertion. The assertion checks the strategy's absolute token balance against `amount` rather than the delta produced by the withdrawal.
`AaveV3ARBUSDCStrategy._deallocate` captures `usdcBalanceBefore` but the subsequent `require` ignores it:

```
1  uint256 usdcBalanceBefore = TokenUtils.safeBalanceOf(address(usdc), address(this)); //
   captured but unused
2  pool.withdraw(address(usdc), amount, address(this));
3  require(TokenUtils.safeBalanceOf(address(usdc), address(this)) >= amount, "Strategy
   balance is less than the amount needed");
```

Because the check compares the absolute post-withdrawal balance to `amount`, any pre-existing idle USDC held by the strategy can satisfy it even if `pool.withdraw` returned fewer tokens than requested.
The identical pattern is present in `AaveV3ARBWETHStrategy._deallocate`, where `wethBalanceBefore` is captured but unused in the subsequent assertion.

Impact

Low.

Recommendation

Replace the absolute balance check with a delta check using the pre-captured snapshot in both strategies:

```
1  // AaveV3ARBUSDCStrategy._deallocate
2  - require(TokenUtils.safeBalanceOf(address(usdc), address(this)) >= amount, "
   Strategy balance is less than the amount needed");
3  + require(TokenUtils.safeBalanceOf(address(usdc), address(this)) >=
   usdcBalanceBefore + amount, "Strategy balance is less than the amount needed");
```

```
// AaveV3ARBWETHStrategy._deallocate
- require(TokenUtils.safeBalanceOf(address(weth), address(this)) >= amount, "
Strategy balance is less than the amount needed");
+ require(TokenUtils.safeBalanceOf(address(weth), address(this)) >=
wethBalanceBefore + amount, "Strategy balance is less than the amount needed");
```

This ensures the check validates that `pool.withdraw` delivered `amount` new tokens, independent of any pre-existing idle balance.

Developer Response

Fixed in commit: 9234c52

2.9 Gas Savings Findings

2.9.1 Redundant post-withdraw balance check in ERC4626 strategies

Technical Details

The same `_deallocate` pattern appears across multiple ERC4626-based strategies:

- `MorphoYearnOGWETH.sol:51`
- `EulerWETHStrategy.sol:35`
- `EulerUSDCStrategy.sol:35`
- `PeapodsETHStrategy.sol:37`
- `PeapodsUSDCStrategy.sol:31`

Each performs:

```
vault.withdraw(amount, address(this), address(this));
require(TokenUtils.safeBalanceOf(address(asset), address(this)) >= amount, "Strategy
balance is less than the amount needed");
```

Under ERC4626 semantics, `withdraw(amount, receiver, owner)` should transfer exactly `amount` underlying assets or revert. For compliant vaults, this post-withdraw `balanceOf >= amount` check is therefore redundant and adds avoidable gas on every deallocation.

Impact

Gas Savings.

Recommendation

Remove the redundant balance check after `vault.withdraw(...)` in `_deallocate`, and keep approval/return flow unchanged.

Developer Response

Fixed in bdb9c9d.

2.10 Informational Findings

2.10.1 Aave V3 MYT strategies can return incorrect `amount`

Technical Details

Calls to Aave's `IPool.withdraw(uint256)` can be supplied with `type(256).max` as a parameter to specify the intention of withdrawing the position's entire aToken balance. In such case, both `_deallocate` implementations for Aave strategies would return `type(uint256).max` instead of the actual amount withdrawn. Furthermore, in such case the slippage check present after the withdrawal call will fail, as it attempts to verify that the contract's token balance is larger than `type(uint256).max`.

Impact

Informational.

Recommendation

Given that `IPool.withdraw` returns the actual amount of assets withdrawn within the call, return such value.

Developer Response

Fixed in 0b7be07.

2.10.2 `_previewAdjustedWithdraw` may overestimate net withdrawal amount due to rounding direction

Technical Description

Several ERC-4626-based strategy implementations override `_previewAdjustedWithdraw` to estimate the amount a user can fully withdraw after accounting for vault fees and slippage. The function is documented as returning a conservative estimate: "the correct amount that can be fully withdrawn, accounting for losses due to slippage, protocol fees, and rounding differences." However, the final slippage deduction rounds in the wrong direction. Taking `EulerWETHStrategy` as a representative example:

```
function _previewAdjustedWithdraw(uint256 amount) internal view override returns (
uint256) {
    uint256 sharesNoFee = vault.convertToShares(amount);
    uint256 sharesWithFee = vault.previewWithdraw(amount);
    uint256 feeShares = sharesWithFee > sharesNoFee ? sharesWithFee - sharesNoFee : 0;
    uint256 feeAssets = vault.convertToAssets(feeShares);
```

```
    uint256 netAssets = amount - feeAssets;
    return netAssets - (netAssets * params.slippageBPS / 10_000); // @audit ret value is rounded up
}
```

The expression `netAssets * params.slippageBPS / 10_000` rounds down due to integer division. Since this value is subtracted from `netAssets`, the final return value is rounded up, producing an estimate that is slightly higher than the true amount recoverable from the vault. The same exact pattern is present in 8 strategy contracts:

- `EulerWETHStrategy`
- `EulerUSDCStrategy`
- `EulerARBWETHStrategy`
- `EulerARBUSDCStrategy`
- `FluidARBUSDCStrategy`
- `MorphoYearnOGWETHStrategy`
- `PeapodsETHStrategy`
- `PeapodsUSDCStrategy`

A similar pattern is also present in the following contracts:

- `AaveV3ARBUSDCStrategy`
- `AaveV3ARBWETHStrategy`

Impact

Informational.

Recommendation

Compute the return value using a single multiplication and floor division:

```
- return netAssets - (netAssets * params.slippageBPS / 10_000);
+ return netAssets * (10_000 - params.slippageBPS) / 10_000;
```

Developer Response

Fixed in da317185.

2.10.3 `MYTStrategy.dexSwap` uses bare `approve` and `transfer` instead of `SafeERC20` wrappers

Technical Description

`MYTStrategy.dexSwap` calls `IERC20.approve` directly on the `from` token to grant the `allowanceHolder` an allowance before executing a DEX swap, and resets it to zero afterwards. Similarly, `withdrawToVault` uses a bare `transfer`.

A small number of ERC-20 tokens do not return a `bool` from `approve` and `transfer`, causing the bare `IERC20` call to revert when the compiler expects a return value. OpenZeppelin's `SafeERC20.safeApprove` and `SafeERC20.safeTransfer` handle missing return values gracefully, and are already used in other parts of the Alchemix V3 codebase.

Impact

Informational.

Recommendation

Import and apply `SafeERC20.safeApprove` and `SafeERC20.safeTransfer` in `MYTStrategy`:

Developer Response

Fixed in commit a209d80

2.10.4 `block.number` on Arbitrum returns L1 block numbers, misaligning `timeToTransmute` with expected block cadence

Technical Description

The `Transmuter` contract uses `block.number` throughout its staking and position-tracking logic to measure transmutation progress in blocks. On Ethereum L1, `block.number` increments once per ~12-second slot, and `timeToTransmute` can be configured to reflect this cadence.
On Arbitrum, however, `block.number` does not return the Arbitrum chain's own block count. Instead, it returns an approximation of the latest L1 block number to which the Arbitrum sequencer has committed. This value increments far less frequently than actual Arbitrum blocks, roughly once per ~12 seconds (tracking Ethereum slots), while Arbitrum itself produces blocks approximately every 0.25 seconds.
The `AlchemistV3` similarly uses `block.number` for earmark windows (`lastEarmarkBlock`, `lastRedemptionBlock`), flash loan guards (`lastMintBlock`, `lastRepayBlock`), and to query the staking graph (`queryGraph(lastEarmarkBlock + 1, block.number)`).
Because Arbitrum's `block.number` can remain constant across multiple Arbitrum transactions within the same L1 block window, the same-block guards (e.g., `CannotRepayOnMintBlock`, `PrematureClaim`) will provide a coarser protection window than on L1 as all transactions within an L1 block interval share the same `block.number`.

Impact

Informational.

Recommendation

Document that `timeToTransmute` must be specified in terms of L1 block increments when deploying on Arbitrum, not Arbitrum-native block counts, in order to avoid a semantic difference which would result in ~48x longer transmutation time.
Alternatively, consider using Arbitrum's `ArbSys(address(100)).arbBlockNumber()` precompile, or Uniswap's blocknumberish, to obtain the true L2 block number when deployed on Arbitrum, ensuring consistent block-level granularity regardless of chain.

Developer Response

Acknowledged.

2.10.5 `AlchemistV3` exceeds Ethereum's contract size limit and cannot be deployed

Technical Details

The compiled `AlchemistV3` contract produces a runtime bytecode of 24,651 bytes under the production build profile (`via_ir = true`, `optimizer_runs = 800`), exceeding the 24,576-byte limit imposed by EIP-170 by 75 bytes. The contract cannot be deployed to mainnet or any chain that enforces this limit.
The contract consolidates all core CDP logic (deposits, withdrawals, minting, repayment, earmarking, liquidation, and administrative functions) into a single contract, which is the root cause of the size violation.

Impact

Informational.

Recommendation

Reduce the contract's bytecode below the EIP-170 limit by removing redundant code and/or breaking the contract into smaller ones.

Developer Response

Fixed in 4b7846.

2.10.6 `AlchemistV3.redeem` contains an unreachable `newIndex == 0` guard

Technical Description

In `AlchemistV3.redeem`, after computing the new redemption weight index via `mulQ128`, a guard checks whether `newIndex` collapsed to zero:

```
if (ratioWanted == 0) {
    newEpoch += 1;
    newIndex = ONE_Q128;
} else {
    newIndex = FixedPointMath.mulQ128(oldIndex, ratioWanted);

    if (newIndex == 0) {
        // dead code
        newEpoch += 1;
        newIndex = ONE_Q128;
    }
}
```

`newIndex` is assigned from `FixedPointMath.mulQ128(oldIndex, ratioWanted)`. Since `mulQ128` returns zero if and only if at least one of its inputs is zero, `newIndex == 0` requires `oldIndex == 0 || ratioWanted == 0`. Both are ruled out by preceding logic:

1. **`oldIndex == 0`**: The normalization block above resets `oldIndex` to `ONE_Q128` whenever `packedOld == 0` or `oldIndex == 0`. By the time execution reaches the `mulQ128` call, `oldIndex` is guaranteed to be nonzero.
2. **`ratioWanted == 0`**: The `ratioWanted == 0` case is handled by the outer `if` branch, which advances the epoch and resets `newIndex` to `ONE_Q128`. The `else` branch containing the `mulQ128` call is only entered when `ratioWanted != 0`.

Since both preconditions for a zero result are excluded, the `newIndex == 0` guard is unreachable.

Impact

Informational. The dead branch cannot be triggered and has no effect on protocol behavior. Its presence reduces the code's readability and maintainability.

Recommendation

Remove the unreachable guard:

```
 } else {
     newIndex = FixedPointMath.mulQ128(oldIndex, ratioWanted);
-
-    if (newIndex == 0) {
-        newEpoch += 1;
-        newIndex = ONE_Q128;
-    }
 }
```

Developer Response

Fixed in 10c6cb.

2.10.7 Unused import

The identifier is imported but never used within the file

Technical Details

```
File: src/AlchemistETHVault.sol

6: import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
```

AlchemistETHVault.sol#L6

```
File: src/AlchemistV3Position.sol

7: import {IAlchemistV3Position} from "./interfaces/IAlchemistV3Position.sol";
```

AlchemistV3Position.sol#L7

```
File: src/Transmuter.sol

8: import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

9: import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";
```

Transmuter.sol#L8, Transmuter.sol#L9

```
File: src/libraries/SafeERC20.sol

6: import {IllegalState} from "../base/ErrorMessages.sol";
```

libraries/SafeERC20.sol#L6

```
File: src/strategies/mainnet/MorphoYearnOGWETH.sol

5: import {IMYTStrategy} from "../../interfaces/IMYTStrategy.sol";
```

strategies/mainnet/MorphoYearnOGWETH.sol#L5

```
File: src/strategies/optimism/MoonwellUSDCStrategy.sol

4: import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
```

strategies/optimism/MoonwellUSDCStrategy.sol#L4

```
File: src/strategies/optimism/MoonwellWETHStrategy.sol

4: import {IERC4626} from "@openzeppelin/contracts/interfaces/IERC4626.sol";
```

strategies/optimism/MoonwellWETHStrategy.sol#L4

Impact

Informational. Improve code quality.

Recommendation

Remove unused imports.

Developer Response

Fixed in 10c6cb.

2.10.8 Unnecessary ternary operator

Technical Details

The ternary operator employed at `AlchemistV3.sol#L805` isn't necessary, given that a previous if-statement at `AlchemistV3.sol#L792`, which leads to an early return, has already verified whether the `debt >= collateral` condition holds or not.

Impact

Informational.

Recommendation

Remove the unnecessary ternary operator:

```
        // fee is taken from surplus = collateral - debt
-       uint256 surplus = collateral > debt ? collateral - debt : 0;
+       uint256 surplus = collateral - debt;

        fee = (surplus * feeBps) / BPS;
```

Developer Response

Fixed in 10c6cb.

2.10.9 MYTStrategy cannot rescue airdrops

Technical Details

`MYTStrategy` currently has no generic token-rescue method for arbitrary ERC20s sent directly to a strategy contract.
There is no owner/admin function to recover unrelated tokens (airdrops).Many adapters also restrict `claimRewards` to specific tokens and flows, so it is not a generic fallback for arbitrary token recovery.
As a result, unexpected ERC20 balances can become permanently stranded in strategy contracts.

Impact

Informational.

Recommendation

Add a restricted owner-only rescue function in `MYTStrategy`, with explicit denylist protections:

1. Allow rescue of arbitrary ERC20 tokens sent by mistake.
2. Disallow rescuing core strategy principal tokens, including:

- `MYT.asset()`
- protocol receipt/share tokens used by the strategy (for example `mToken`, `aToken`, ERC4626 vault shares, staking receipt tokens)
- vault tokens used by the strategy's underlying protocol path (stETH for the WStethStrategy).

Developer Response

addressed in 486160.

2.10.10 `Transmuter.queryGraph` manual ceiling division can be replaced with `mulDivUp`

Technical Description

`Transmuter.queryGraph` computes a ceiling division manually:

```
return (queried / BLOCK_SCALING_FACTOR).toUint256() + (queried % BLOCK_SCALING_FACTOR ==
 0 ? 0 : 1);
```

The expression is equivalent to `FixedPointMath.mulDivUp(queried.toUint256(), 1, uint256(BLOCK_SCALING_FACTOR))`, a helper that is already used consistently throughout the codebase for ceiling division. The manual form requires the reader to parse the modulo remainder check before recognising the operation as a simple ceiling division, reducing readability.

Impact

Informational.

Recommendation

Replace the manual ceiling division with a call to `FixedPointMath.mulDivUp`:

```
-         return (queried / BLOCK_SCALING_FACTOR).toUint256() + (queried %
BLOCK_SCALING_FACTOR == 0 ? 0 : 1);
+         return FixedPointMath.mulDivUp(queried.toUint256(), 1, uint256(
BLOCK_SCALING_FACTOR));
```

Developer Response

Addressed in 10c6cb.

2.10.11 Incorrect addresses commented in `MoonwellUSDCStrategy`

Technical Details

`MoonwellUSDCStrategy` defines the `mUSDC` and `usdc` immutables, with the following comments next to their definition:

```
IMToken public immutable mUSDC; // Moonwell market mUSDC (mToken) 0
xd0670AEe3698F66e2D4dAf071EB9c690d978BFA8
IERC20 public immutable usdc; // 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
```

Such comments both indicate incorrect addresses: 1. `0xd0670AEe3698F66e2D4dAf071EB9c690d978BFA8` is the address for mUSDC on Moonriver (link) 2. `0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48` is the address for USDC on Ethereum (link)

Impact

Informational.

Recommendation

Remove the incorrect comments or insert the correct addresses for Optimism.

Developer Response

addressed in a72369.

2.10.12 Magic value constants used to populate `permissionedCalls`

Technical Details

The constructors of `AlchemistAllocator` and `AlchemistCurator` populate the `permissionedCalls` mapping using hardcoded hexadecimal constants:

```
// AlchemistAllocator constructor
permissionedCalls[0x5c9ce04d] = true; // allocate(address adapter, bytes memory data,
uint256 assets)
permissionedCalls[0x4b219d16] = true; // deallocate(address adapter, bytes memory data,
uint256 assets)

// AlchemistCurator constructor
permissionedCalls[0xf6f98fd5] = true; // increaseAbsoluteCap(bytes memory, uint256)
permissionedCalls[0x8c54519b] = true; // decreaseAbsoluteCap(bytes memory, uint256)
permissionedCalls[0x2438525b] = true; // increaseRelativeCap(bytes memory idData,
uint256 newRelativeCap)
permissionedCalls[0x57975270] = true; // decreaseRelativeCap(bytes memory idData,
uint256 newRelativeCap)
permissionedCalls[0xb192a84a] = true; // setIsAllocator(address account, bool
newIsAllocator)
```

While inline comments document the intended function signatures, the use of magic values makes the code harder to read and verify. If a function signature were to change, the hardcoded selector would silently become stale without producing a visible error.

Impact

Informational.

Recommendation

Replace the hardcoded hex constants with `<Interface>.<function>.selector` expressions. For example, within `AlchemistAllocator`:

```
-        // allocate(address adapter, bytes memory data, uint256 assets)
-        permissionedCalls[0x5c9ce04d] = true;
-        // deallocate(address adapter, bytes memory data, uint256 assets)
-        permissionedCalls[0x4b219d16] = true;
+        permissionedCalls[IVaultV2.allocate.selector] = true;
+        permissionedCalls[IVaultV2.deallocate.selector] = true;
```

Developer Response

Code has been removed with a switch to the whitelist approach on b273fc.

2.10.13 Duplicated strategy implementations

Technical Details

The following sets of contracts are near-identical and only differ by token/vault wiring and naming:

- ERC4626-simple pattern (same `_allocate`, `_deallocate`, `_totalValue`, `_previewAdjustedWithdraw`): src/strategies/mainnet/EulerUSDCStrategy.sol
  src/strategies/mainnet/EulerWETHStrategy.sol
  src/strategies/arbitrum/EulerARBUSDCStrategy.sol
  src/strategies/arbitrum/EulerARBWETHStrategy.sol
  src/strategies/mainnet/PeapodsUSDCStrategy.sol
  src/strategies/mainnet/PeapodsETHStrategy.sol
  src/strategies/arbitrum/FluidARBUSDCStrategy.sol
  src/strategies/mainnet/MorphoYearnOGWETH.sol (very close variant)
- Aave pattern (same supply/withdraw/value/reward flow):
  src/strategies/arbitrum/AaveV3ARBUSDCStrategy.sol
  src/strategies/arbitrum/AaveV3ARBWETHStrategy.sol
  src/strategies/optimism/AaveV3OPUSDCStrategy.sol
- Moonwell pattern (same mToken flow; WETH version adds ETH wrap):
  src/strategies/optimism/MoonwellUSDCStrategy.sol
  src/strategies/optimism/MoonwellWETHStrategy.sol
- Tokemak pattern (same stake/redeem/reward flow with asset-specific wiring):
  src/strategies/mainnet/TokeAutoUSDStrategy.sol
  src/strategies/mainnet/TokeAutoETHStrategy.sol

Impact

Informational. Increases maintenance cost and review burden.

Recommendation

Refactor into shared logic per family.

Developer Response

Fixed different strategies in the following commits: - erc4626 : ad8efc9 - aave : a689ed4 - moonwell : f6a308a - toke : affde43, most recent commit : dc26088

2.10.14 Unused variable

Technical Details

The following variables aren't used: - `pendingStrategy` - `router` : 1 and 2 - `unsteth` - `estApr` and `estApy` - `usdcBalanceBefore` - `wethBalanceBefore` - `MIN_SNAPSHOT_INTERVAL` - `lastSnapshotTime` - `lastIndex` - `METATXN_VIP`, `CURVE_TRICRYPTO_VIP`, `UNISWAPV4_VIP`, `DODOV1_VIP` and `DODOV2_VIP` in `ZeroXSwapVerifier` - `lastRedemptionBlock` : this storage variable is only written to within `AlchemistV3.redeem`

Impact

Informational

Recommendation

Remove the unused variables.

Developer Response

addressed in d69f55c.

2.11 Final Remarks

The review identified no critical issues, but it did uncover 3 high-severity and 5 medium-severity. The Alchemix team was responsive during the engagement and addressed the substantial majority of findings in follow-up commits, which materially improved the final security posture. Even so, the protocol remains complex and operationally sensitive.