Audit Report

Alchemix v2

Delivered: 2022-01-13

Prepared for Alchemix by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. has audited the smart contract source code for Alchemix v2. The review was conducted from 2021-11-01 to 2021-12-17.

Alchemix engaged Runtime Verification in checking the security of their Alchemix v2, which is an upgrade of their v1 in that it allows multiple underlying tokens per synthetic asset, as well as multiple yield tokens, and that it allows more DeFi integration by allowing interactions from contracts, not just externally owned accounts (EOAs). It also uses a more efficient Transmuter algorithm and a modified way for the Transmuter to boost the yields of other users. Alchemix started engaging RV in auditing their v2 in June 2021, and RV has assisted in the ongoing design of the system through two separate engagements. This is the second engagement which reviews the system in its entirety, in its intended final form.

The issues which have been identified can be found in section Findings. A number of additional suggestions have also been made, and can be found in Informative findings. We have also verified a number of security properties, which can be found under Checked properties.

We have found that the Alchemix team follows best practices, is mindful in introducing features and keeps security a high priority throughout the development process. A protocol managing investments always carries some amount of risk, and much effort has been spent finding minimal trade-offs between performance, usability, simplicity and robustness. The test suite and documentation are satisfactory, and the deployment process mechanized enough that it can sensibly be tested.

Scope

The audited smart contracts are:

- AlchemistV2
- AlchemicToken
- TransmuterV2
- TransmuterBuffer
- WETHGateway
- YearnTokenAdapter

In the process, the following libraries were audited

- Limiters
- LiquidityMath
- ContractWhitelister
- Sets

- FixedPointMath
- SafeCast
- Tick

The audit has focused on the above smart contracts and libraries, and has assumed correctness of external contracts and libraries they make use of. The libraries are widely used and assumed secure and functionally correct.

The review encompassed a private code repository, which contains a Hardhat project with test scripts. The code was frozen for review at commit `b5f677106d31aa5301efadaaf542e07b64c29531`.

The review is limited in scope to consider only contract code. There was no off-chain and client-side code present in the repository. Apart from the contract code, the repository contains several deployment scripts. These have not been formally audited and were not expected to be frozen, although we have used them to understand and verify the correct deployment flow, and we have found nothing noteworthy in those scripts we have considered.

Assumptions

The audit is based on the following assumptions and trust model.

1. The admin addresses of the respective contracts need to be absolutely trusted. Due to the upgradability of the Alchemist, Transmuters and TransmuterBuffer, the governance address(es) which control these contracts have the ability to deploy new code which could withdraw any tokens in those contracts. They have the ability to take all yield tokens out of an Alchemist, and to take all underlying tokens out of the TransmuterBuffer (see Appendix 1: Token types per entity for a table describing which addresses control which types of tokens). Governance at the time of launch will be a multisig controlled by the Alchemix team, and a migration to a DAO-based model is planned.
2. The keepers of the Alchemist and TransmuterBuffer also need significant trust. They have the ability to trigger significant token movements. In the case of the Alchemist keepers, they have the ability to trigger harvests, which gives them power to claim the gains of CDP holders (see B12: Keepers can steal entire harvest).
3. We assume that the deployers and the governance address take relevant steps to ensure that the state of the deployed contracts remains correct. For example, the TransmuterBuffer needs to keep its listed of supported tokens up to date with that in the Alchemist; and setting the `transmuter` of an Alchemist to the wrong address would lead to harvests being lost.
4. In addition to setting *correct* state, it is also contingent upon governance to maintain *reasonable* state. This includes only accepting trustworthy yield tokens, setting protocol,

max loss numbers, debt ceiling and allowed yield token balances to reflect the risk profiles of the underlying yield tokens.

Note that the assumptions roughly assume "honesty and competence". However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Thirdly, we discussed the most catastrophic outcomes with the team, and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with the Alchemix team to discuss our findings and together come up with mitigations and reasonable trade-offs.

This report describes, in Contract Description and Invariants, the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found in Findings, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter in Informative findings. Finally, we also give an overview of the important security properties we and invariants we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Contract Description and Invariants

This section describes the contracts at a high-level, and which invariants of its state we expect it to always respect at the end of a contract interaction.

AlchemistV2

The Alchemist allows users to create a collateralized debt position (CDP) by depositing tokens as collateral and then taking debt against the collateral by minting debt tokens. The ratio of underlying collateral to debt must maintain a minimum value set as a parameter of the contract. Underlying tokens deposited into the CDP are put into a yield farm in exchange for yield tokens. The user selects which yield tokens should be put in their CDP from a set of allowed ones. The yield tokens appreciate over time, and the yield can then be harvested to pay the users' debts.

Multiple independent versions of the Alchemist contract will be deployed, each supporting a different set of 1:1 stablecoins as underlying tokens. Each Alchemist also has a different debt token pegged to the price of the underlying assets.

Operations

Users of the Alchemist have access to the following operations:

- Depositing (underlying or yield) tokens into a CDP.
- Withdrawing (underlying or yield) tokens from a CDP.
- Minting debt tokens from a CDP.
- Burning debt tokens to pay off debt.
- Repaying debt using underlying tokens.
- Liquidating collateral to pay off debt.
- Donating credit to boost the yield of other users by burning debt tokens.
- Approving a third party to withdraw or mint from their CDP.
- Poke a CDP, updating its debt based on the accrued yield.

Users can deposit and pay back debt (either with underlying tokens or by burning debt tokens) in any CDP, and withdraw and mint either from their own CDP or a CDP for which they have been given an allowance to do so. Anyone can poke a CDP to update its debt. In addition to externally-owned accounts, the Alchemist keeps a whitelist of contracts that are allowed to interact with it in this way (except for poking, which can be done even by a non-whitelisted contract).

In addition, the Alchemist contract has three types of privileged roles[1]: admin, sentinels and keepers. The address designated as the admin can perform the following operations:

- Add new tokens to the list of underlying and yield tokens supported by the Alchemist.
- Enable and disable tokens from this list.
- Add and remove sentinels and keepers.
- Transfer the admin role to a different address (must be accepted by the new admin).
- Configure the Alchemist's parameters, including limits, fees and the addresses of the transmuter and protocol fee receiver.
- Reset (“snap”) the expected value of a yield token to the current value. Since deposits, withdrawals and liquidations are blocked if the value of a yield token suddenly drops significantly below its expected value, this can prevent the contract from becoming unusable if the yield token doesn’t recover, or takes too long to recover.

Sentinels can disable underlying and yield tokens, while keepers can trigger harvests of the yield tokens.

When a keeper harvests a yield token, the Alchemist calculates how much yield has been generated since the last harvest. If this amount is positive, then it is converted from yield tokens to the underlying token. After taking out a fee, which is sent to the address designated as the fee receiver, the remaining amount is sent to the `transmuter` address, which may be a TransmuterBuffer. At the same time, the total debt in the Alchemist decreases by the same amount sent to the Transmuter, with each CDP receiving credit proportional to their balance. This corresponds to debt tokens that were previously backed by the debt in the CDPs now being backed instead by the underlying tokens sent to the Transmuter. Whenever a user repays debt using underlying tokens or liquidates their shares in the CDP, the underlying tokens are also sent to the Transmuter.

Internal bookkeeping

The internal bookkeeping of the Alchemist is done using different variables:

- Every yield token has a `balance`, `totalShares`, `expectedValue` and `accruedWeight`.
- Every CDP has a `balance` and `lastAccruedWeight` for each yield token.

The `balance` of a yield token is the number of tokens of that type stored in the Alchemist. Every CDP owns a number of shares of that balance, and the `totalShares` variable stores the total number of shares. The `balance` of a CDP is measured in shares of the yield token, rather than in units of the token itself. Therefore, the actual balance of a CDP in yield tokens is calculated by

[1] Note that the Alchemist deployment process has been changed, and at deployment it will use an upgradeable proxy. This means that apart from the contract privileged roles, there will be a role of proxy admin, which can upgrade the code of the Alchemist address.

$b \cdot s / t$, where b is the yield token's `balance`, s is the number of shares in the CDP (given by the CDP's `balance`) and t is the total number of shares (given by the `totalShares` variable).

The reason to keep track separately of `balance` and `totalShares` is that when a harvest occurs the `balance` of the yield token decreases, as a number of yield tokens corresponding to the harvested amount is converted to underlying tokens and sent to the TransmuterBuffer. However, each CDP still owns the same number of shares of the remaining amount. After harvest, each share corresponds to a smaller number of yield tokens, but is still worth the same in underlying tokens as when the share was issued, since the yield token has appreciated since then.

The `expectedValue` of a yield token is the value of its `balance` in underlying tokens ignoring any yield that they have accrued since being deposited. In other words, whenever yield tokens are deposited into the Alchemist, their value at time of deposit is added to `expectedValue`. Similarly, when yield tokens are withdrawn or liquidated, their value at time of withdrawal/liquidation is subtracted from `expectedValue`. When the yield token is harvested, the current value of its balance in underlying tokens is calculated, and any amount above the `expectedValue` is harvested as yield and used to pay back debt in the CDPs.

The harvested yield is not immediately credited to each CDP at harvest. Instead, it is used to increase the `accruedWeight` of the yield token. This value starts at 0 and increases every time the token is harvested. The value of the increase is equal to a / t, where a is the harvested amount in underlying tokens (minus a fee) and t is the total number of shares of the yield token. Therefore, the accrued weight represents how much yield each share has accrued through multiple harvests.

The `lastAccruedWeight` of a CDP is the last weight that has been used to update the CDP's debt (for a specific yield token). When a CDP is poked, the weight that has been accrued since the last update is calculated by `accruedWeight - lastAccruedWeight`. This value is then multiplied by the number of shares of the CDP and subtracted from the CDP's debt. In this way, the debt of the CDP decreases by the value that its shares have appreciated by since the last update. The `lastAccruedWeight` is then updated to the current `accruedWeight`.

It's worth noting that the `accruedWeight` of a yield token can also be increased by the `donate` function. This function allows a user to burn debt tokens to donate credit to all other CDPs that have shares of the yield token. This is done by adding $a / (t - s)$ to the `accruedWeight`, where a is the amount of debt tokens burned, t is the total number of shares of the yield token, and s is the number of shares in the user's CDP. This is effectively distributing the amount burned across all shares of the yield token except those owned by the user. To prevent the shares owned by the user from also receiving the donated weight when the user's CDP is next poked, the user's `lastAccruedWeight` is updated to this new `accruedWeight`.

Pre-harvest

PR #49 introduces a pre-harvest operation that is performed before critical functions of the Alchemist such as deposits, withdrawals, minting and liquidations. This operation performs only the first step of the harvest, consisting in calculating the yield earned by the yield tokens and removing a corresponding amount of yield tokens from the balance. However, instead of immediately distributing the yield as credit to the CDPs and transferring the removed yield tokens to the Transmuter, the tokens are instead stored in a `harvestBuffer`, to then be distributed and transferred at the next full harvest.

Pre-harvest prevents issues caused by users withdrawing or minting not only from the deposited collateral, but also from yield earned by yield tokens, which can then lead to bookkeeping errors and under-collateralization when this yield is harvested (see A04). By pre-harvesting, the contract ensures that the yield is taken away from the balance before the user can operate on it.

Invariants

The invariants below all model the Alchemist using real (as opposed to integral) math, i.e. assumes no rounding errors. Section C05 reasons formally about the code in order to build confidence that they hold in the current implementation. The version of the code considered includes the pre-harvest operation as implemented in PR #49.

Invariant A1: Assume all CDPs are fully updated (using `_poke`). Let m be the amount of debt tokens minted by the Alchemist, b the amount of debt tokens burned by the Alchemist, d the sum of all debts in the Alchemist, and t the amount of underlying tokens sent to the TransmuterBuffer from the Alchemist. Then, $m = b + d + t$. Note that if a CDP has credit (negative debt) this amount is *subtracted* from d.

Invariant A2: The total number of shares of a yield token is equal to the sum of the shares of that yield token over all CDPs.

Invariant A3: Let b be the balance and t the total number of shares of a given yield token. Then, $b \leq t$, and $b = 0$ if and only if $t = 0$[2].

Invariant A4: Unless the token has suffered a loss, every operation that changes the balance or expected value of a yield token leaves the expected value equal to the current value, calculated by multiplying the price of the token by its balance.

Invariant A5: Assume the `AddressSet` data structure behaves correctly[3]. Then, a user has balance in a token if and only if that token is in the set of deposited tokens for that user.

[2] This invariant is dependent on the fix to A04.
[3] This assumption is dependent on the fix to A02.

Invariant A6: Liquidating some amount of a debt will have the exact same result as repaying that amount of debt after withdrawing the same amount of collateral.

Invariant A7: Assuming the price of a yield token never drops to 0, the expected value of the yield token equals 0 only if its balance equals 0[4].

The invariants below depend on the additional assumption that governance will not change parameters in such a way that they are violated. This includes disabling a yield/underlying token when there is a balance of that token in a CDP or increasing the minimum collateralization ratio above the current ratio of an existing CDP. These assumptions are not enforced in the code, although safeguards could be added to prevent them, for example checking that the balance of a yield token is 0 before disabling it or only allowing the minimum collateralization ratio to be decreased, but not increased.

Invariant A8: If a yield token or its underlying token is disabled in the protocol, then no user has any balance in that yield token.

Invariant A9: Assume no loss occurs on yield tokens. Then, every CDP (after being updated by `_poke`) is always "healthy", meaning it maintains at least the minimum collateralization ratio (assuming this ratio is at least 1)[5].

[4] This invariant is dependent on the fix to A04.
[5] This invariant is dependent on the fix to A04.

Limiters

Limiters are used to limit the rate of certain operations in the Alchemist. The Alchemist has a single minting limiter, which limits how many debt tokens can be minted over a period of time, and a repay and liquidation limiter for each underlying token, which limit how many tokens can be spent in repayments or liquidations over a period of time.

Each limiter provides an upper limit on the number of tokens that can be used for the associated operation. The remaining amount of tokens that can still be used can be obtained using the `get` function. When an operation is performed, this amount should be decreased by calling the `decrease` function, passing the number of tokens that have been used. For example, when `amount` debt tokens are minted, the Alchemist calls `_mintingLimiter.decrease(amount)`. If this amount surpasses the remaining amount allowed by the limiter, `decrease` reverts.

The remaining amount allowed by the limiter increases back over time, up to the maximum. The rate of this increase is given in tokens/block, and calculated when the limiter is constructed based on how many blocks it should take for the limiter to return to the maximum from 0. The `get` function calculates the current value of the limiter on the fly using the formula

$$\min(\mathit{maximum}, \mathit{lastValue} + (\mathit{currentBlock} - \mathit{lastBlock}) \cdot \mathit{rate})$$

where *maximum* is the limiter's upper limit, *lastValue* is the value after the last decrease, *currentBlock* is the current block (in which `get` is called), *lastBlock* is the block at the last decrease, and *rate* is the rate of increase. There is also an `update` function that updates *lastValue* to the current value calculated by `get` and *lastBlock* to the current block. This is equivalent to calling `decrease(0)`.

AlchemicToken

The AlchemicToken is the ERC20 token contract used for the synthetic tokens (alTokens) minted as debt tokens by the Alchemist. Each Alchemist has its own alToken. The different Alchemists should be added by governance to the whitelist of the appropriate token, in order to be able to mint them.

The new version of the AlchemicToken has two main differences to the original AlToken contract. The first is the removal of the blacklist, which allowed sentinels to permanently block an Alchemist contract from minting tokens. This opened the possibility of, in the case of a sentinel account being compromised, an attacker being able to render an Alchemist unusable. The current version of AlchemicToken is limited to pausing an Alchemist contract, which allows them to be unpaused later. The second difference is the introduction of flash minting. A fee, configurable by governance, is charged for every flash mint. While each Alchemist has a ceiling of how many tokens it can mint, flash minting can mint any number of tokens as long as the amount flash minted plus the amount of tokens already in circulation is within the representation range of a `uint256`.

The AlchemicToken contract uses the `totalMinted` mapping to keep a count of how many tokens have been minted by each Alchemist, in order to ensure that they don't surpass the Alchemist's mint ceiling. This count increases automatically whenever new tokens are minted, but needs to be decreased manually when tokens are burned by calling `lowerHasMinted`. Therefore, the Alchemist must always call `lowerHasMinted` after calling `burn` or `burnFrom` in the AlchemicToken. Since there is no way for a general user to lower `totalMinted`, alTokens should only be burned by an Alchemist contract (see B15).

The new AlchemicToken contract will be used for the new synthetic tokens introduced in the new version of the protocol, but alTokens already in circulation (alUSD and alETH) will continue to use the original AlToken contract. This means that some V2 Alchemists will use AlchemicToken as the debt token, while others will use AlToken. Other than the differences listed above, the two versions of the token contract have the same functionality and function signatures. Since the Alchemist does not make use of flash minting, it is compatible with both. Note, however, that the Alchemist for an alToken already in circulation can still be blacklisted if a sentinel account is compromised.

Transmuter and TransmuterBuffer

These contracts are controlled by an admin account set up at initialization time (though their ownership address can change). They are currently undeployed.

Transmuter[6]

The goal of the Transmuter V2 compared to V1[7] is to eliminate the need for forced transmutations. A forced transmutation is when a position in the Transmuter is over-allocated. The user can never withdraw more of the underlying asset (for example, DAI) than the amount of alAssets (for example alUSD) than they have deposited. Still, the Transmuter V1 could credit an unlimited amount of underlying tokens to any position. This was a simple and gas-efficient way to implement the Transmuter, but it means that to unlock the over-allocated funds, users have to force transmute an over-allocated position, essentially closing the over-allocated position and sending the underlying owed to the owner, and taking any over-allocation for themselves, speeding up their own transmutation. The client UI displays positions available for force-transmutation. This then becomes a highly manual process for users, and makes it unsuitable for on-chain integrations.

The different iterations of the Tansmuter V2 all address this issue in different ways. However, each approach comes with trade-offs.

Each position in the Transmuter V2 can be said to have two conceptual balances: the *exchanged* balance and the *unexchanged* balance. The exchanged balance is the amount the user can withdraw in the underlying asset. The unexchanged balance is the amount that they can withdraw in alAssets. When created, the position will be 100% unexchanged, and in time will move to being 100% exchanged, as yield from the yield farms is used to exchange the unexchanged balance for exchanged balance.

The Transmuter's bookkeeping relies on "ticks", a structure of points which everyone who deposited between the same exchange events inhabits. Any users that deposited between the same two exchange events should be exchanged at exactly the same rate.

The Transmuter has three main entry points for users: `deposit` (alTokens), `withdraw` (alTokens), and `claim` (underlying tokens).

[6] This section is lifted from a previous design review report that RV conducted of the Alchemix v2 system, which has not yet been released.
[7] Currently deployed at 0xee69bd81bd056339368c97c4b2837b4dc4b796e7 for alUSD/DAI, and at 0x9fd9946e526357b35d95bcb4b388614be4cfd4ac for alETH/ETH. A web3 frontend maintained by the Alchemix team can be found here: https://app.alchemix.fi/transmute.

Invariant: the sum of all deposits of a single account must always be larger or equal to the sum of withdrawals from that account and the claims.

At any point, the user has an exchanged and unexchanged balance. They can withdraw from the unexchanged balance, and claim from the exchanged balance. Both these balances are always non-negative. On a regular basis, funds are sent from the Alchemist into the Transmuter, triggering the exchange function, which distributes the sent amount fairly[8] among those waiting to be transmuted.

Invariant: an account's exchanged balance plus unexchanged balance must be equal to their total deposits minus their withdrawals and claims.

The Transmuter is not always the best way to convert between alTokens and underlying tokens, since stablecoin markets may offer a better rate and can perform the swap immediately. The Transmuter must therefore also be able to maintain a buffer of funds for the situations when there are more underlying tokens flowing into the Transmuter than there are unexchanged balances.

Invariant: Whenever an exchange event occurs, the total amount of underlying tokens that the Transmuter owns is equal to the buffer, plus the sum of all the exchanged balances.

Invariant: The sum of all exchange event amounts minus the buffer is equal to the sum of all claimed amounts plus the sum of all exchanged balances.

TransmuterBuffer

Each Alchemist has a single TransmuterBuffer for all its underlying tokens. The buffer manages underlying tokens for all the Transmuters for that Alchemist. Each TransmuterBuffer one Transmuter per supported underlying token

Whenever the Alchemist's harvest, or receives repayments or liquidations, the underlying tokens in the control of the protocol increases. These funds are sent to the TransmuterBuffer. The buffer ensures that even if the inflow of underlying tokens is large, that the rate of transmutation is kept steady. The TransmuterBuffer calls `exchange()` on the Transmuter of that token, updating the available claims of all currently transmuting positions.

The TransmuterBuffer maintains a variable, `flowAvailable`, for each underlying token which represents how much of it should be made available to the token's Transmuter. This variable value grows linearly, at a rate set by governance. Governance can not control the variable directly, only its (positive) rate of change.

[8] What is considered "fair" has been a central point of discussion, which we will return to later. For now, suffice to say that a fair distribution means every account is transmuted at the same rate relative to either their unexchanged balance, or some combination of their unexchanged and exchanged balances.

At any given time, the amount of underlying tokens available to a Transmuter to withdraw is the minimum of the `flowAvailable` for that token, and the actual amount of that token controlled by the TransmuterBuffer. Put another way, the Transmuter can only withdraw an amount of tokens if that amount is less than (or equal) to both the `flowAvailable` and the actual amount of buffered tokens. Whenever the Transmuter withdraws funds, the variable decreases by the amount of underlying tokens withdrawn.

The Alchemist sends the tokens directly to the TransmuterBuffer (the actual tokens are transferred), then calls `onERC20Received()` which causes the buffer to update its internal accounting, updating its available flow and its buffered token balance. The TransmuterBuffer in turn calls the Transmuter's `exchange()` function to have it update its internal accounting.

The tokens in the TransmuterBuffer can be deposited to the Alchemist by keepers. The Transmuter has a CDP in the Alchemist which it deposits to. The TransmuterBuffer deposits the underlying tokens in the Alchemist. It has a weight assigned to each yield token and deposits the underlying tokens in different strategies according to their weight. The weights are set by governance and represent how much of the received tokens should be allotted to each strategy in the future, it is not a targeted portfolio balancing.

The TransmuterBuffer is not intended to earn any yield, but instead should only boost the yields of other users. It accomplishes this by calling the *donate* function on the Alchemist, giving up its yield. This call is triggered by keepers. Since the TransmuterBuffer internally keeps track of its underlying balance but not its credit, the credited funds are never available to the Transmuters.

Invariant: the weighting of an underlying token only contains yield tokens that are supported by the Alchemist, and belong to that underlying token.

Invariant: The total exchanged amount in a Transmuter is always less than or equal to the total flowAvailable for the underlying token in the corresponding TransmuterBuffer.

Contract Whitelister

Alchemix v1 did not support any interactions from deployed smart contracts (it checked for correspondence between `tx.origin` and `msg.sender`). This was a precautionary measure to prevent any “DeFi lego” attack, effectively preventing flash loans and integrations of CDPs into other protocols. Instead, only the minted tokens, alUSD and alETH, could be used by external contracts.

With v2 a major goal is to allow interaction with other smart contracts. As a precaution, though, there is support for “whitelisting” in the Alchemist and the Transmuters. When the whitelist is active, the contract will reject any user interaction from any outside contract which is not expressly whitelisted. When the whitelist is inactive, there is no restriction on which addresses may interact as users.

Whether the whitelist is active, and which contracts are whitelisted, is controlled by governance.

WETHGateway

The WETHGateway contract is a utility for using an Alchemist with ether directly, even though the Alchemist only supports ERC20 tokens.

The gateway allows depositing ethers into the Alchemist by first wrapping them into WETH, then depositing them on behalf of the specified receiver. It also allows withdrawing in a similar manner, first unwrapping the WETH and sending ethers to the recipient.

The gateway is designed to work with all Alchemists. It has no dependency on any specific one.

Invariant: The WETHGateway always has an ETH balance of 0 before and after each call, and can only have an ether balance if it was sent ethers from a `selfdestruct` operation or as a minter/validator coinbase.

Even if the WETHGateway contains ethers before a call, it will not cause griefing, though it will cause those ethers to be stuck in the contract.

If the address gets sent WETH, any address can withdraw those WETH as ethers to themselves by calling `withdrawUnderlying`.

YearnTokenAdapter

The YearnTokenAdapter is used as the adapter for Yearn tokens added to the Alchemist, providing functionality to allow the Alchemist to interact with Yearn Vaults by depositing and withdrawing underlying tokens.

The `wrap` function transfers an amount of underlying tokens from the caller (in this case, the Alchemist) to the adapter, which then deposits them in the yearn vault, transferring the corresponding yearn tokens to the recipient (again the Alchemist in this case).

The `unwrap` function conversely transfers Yearn tokens to the adapter and withdraws the corresponding amount in underlying tokens from the vault to the recipient (which may be the Alchemist or another address). The balance in the vault is checked before and after withdrawal to make sure that the correct amount in yearn tokens has been withdrawn. If that is not the case, the function reverts. The `unwrap` function also requires a `maxLoss` parameter to be passed in its `data` argument, in order to prevent withdrawals from the vault if the loss is greater than expected.

The `price` function returns the current price the yearn token is worth, which should correspond to the price used in deposits and withdrawals. The below invariants assume that the `price` variable was obtained by calling `adapter.price()` immediately before.

Invariant: `price * adapter.wrap(amount, recipient, data) == amount`

Invariant: `price * amount == adapter.unwrap(amount, recipient, data)`

The YearnTokenAdapter also provides `defaultWrapData` and `defaultUnWrapData` functions, which can be passed as the `data` argument to `wrap` and `unwrap` respectively. Since `wrap` doesn't make use of the data, `defaultWrapData` simply returns `abi.encode(0x0)`, while `defaultUnWrapData` returns data with `maxLoss` set to `0`, meaning that any loss is unacceptable.

Unlike the previously-audited version of the YieldTokenAdapter, this version does not include a vault registry. Instead, the adapter points to a single vault, which at the time of deployment should be the most recent one. In the future, when it is desired to migrate to a more recent vault, a separate migration contract[9] will be provided, which is out of scope for this audit.

[9] The migration would need to involve deploying a new version of the YieldTokenAdapter using the new vault, adding a new version of the yield token in the Alchemist using the new adapter, and disabling the previous version of the yield token. In order to migrate funds deposited in the Alchemist as the old token, the migration contract would likely need to flash mint tokens to temporarily pay off the user's debts so it can withdraw the funds and migrate them to the new token. After returning the new tokens to the CDP, new debt tokens can be minted to return the flash loan. This would require the user's approval for the contract to withdraw and mint tokens from their CDP.

Findings

This section contains those discoveries from the audit which we expect to be addressed before release. They are listed in no particular order. All have been addressed. If they were addressed in another way than by adopting our recommendation, we elaborate on how the finding was addressed under the “Status” section of each finding.

A01: Whitelisting allows admins to block withdrawals

[Severity: Medium | Difficulty: Low | Category: Security]

The admins can at any time disable or enable whitelisting. If a contract is not whitelisted, but has an Alchemist or Transmuter deposit, it can at any point be blocked from withdrawing. This is counter to the goal of Alchemix to always allow users to control their funds, and requires a great deal of trust in the admins. The only way to completely and forever turn off the whitelist is to renounce ownership over the contract. However, this is not feasible since the owner has other responsibilities and functions in the Alchemist and Transmuter. Thus, the deposits would be forever under too great control by the admin.

Scenario

After some time of operation, the admins turn off whitelisting, allowing full integration by any smart contract. A small DeFi platform starts integrating with Alchemix. Users using contract-based wallets and solutions like Gnosis Safe start using Alchemix. Then, due to a perceived risk, the Alchemix admins turn the whitelist back on. The small platform and the end users are now blocked from withdrawing funds, or interacting at all with the contracts in any way, and are at the mercy of the admins.

Recommendation

We generally support the precaution of an initial whitelist. For the purpose of decentralization and minimized trust in a group of admins or a DAO, we would however recommend that a function is added to the ContractWhitelister to *permanently* turn off the whitelist. This allows strong control and rapid correction during battle testing, while allowing the Alchemist to become more trustless in the future, when deemed ready.

Status

When this issue was raised the assumption was that governance should be imbued with limited trust from users, and should never be able to block withdrawals, for example.

The Alchemix team has decided to make the Alchemist and Transmuter contracts upgradeable. This is not uncommon in DeFi. It does, however, require users to put far more trust in governance honesty and competence.

Either way, this change makes the trust assumptions about governance strictly stronger than what the recommended feature would warrant, and it is thus unnecessary. Nevertheless, the whitelist implementation has now removed the ability for governance to re-enable the whitelist once it has been deactivated. The Alchemist and Transmuter start with an enabled whitelist, and can only deactivate it, never reactivate it.

A02: Incorrect update when removing `AddressSet` element

[Severity: High | Difficulty: Low | Category: Functional Correctness]

In the `Sets` library, the `AddressSet` structure keeps a dynamic array `values`, representing the list of elements in the set, and a mapping `indexes`, which maps an element to its 1-based index. This means that `AddressSet` has the following invariant, for all `0 <= i < values.length`:

```
indexes[values[i]] == i + 1
```

Using 1-based indexing for `indexes` makes sense in order to be able to easily test if a value `v` is not in the set, in which case `indexes[v] == 0`. The above invariant is enforced in the `add` function and assumed in the `remove` function. However, when swapping the element to be removed with the last element of the `values` array, the `remove` function incorrectly sets the new index of the last element as `index` rather than `index + 1`. This can lead to incorrect behavior of the `AddressSet` structure.

Scenario

Suppose a user has yield tokens A, B and C deposited in their CDP. This information is stored as an `AddressSet` called `depositedTokens`. Suppose that `depositedTokens.values = [A, B, C]`, in that order. The following sequence of events then happens:

1. The user withdraws their entire balance of token B. Since the balance of token B is now `0`, the `_burnShares` function calls `depositedTokens.remove(B)`. In the process of removing B, token C gets moved to the second position in `values`, but `indexes[C]` is updated to `1` instead of `2`.
2. The user withdraws their entire balance of token C. Like before, `_burnShares` calls `depositedTokens.remove(C)`. However, since `indexes[C]` is currently set to `1` (the 1-based index of A), token A gets removed instead.
3. The keepers harvest token A.
4. The user performs an operation and their debt is updated by the `_poke` function. Since token A is no longer in `depositedTokens`, it is skipped over by `_poke` and the user does not receive credit from the yield harvested from token A.

Recommendation

Change `self.indexes[lastValue] = index` to `self.indexes[lastValue] = index + 1` in line 49 of `Sets.sol`.

Status

Addressed in PR #31.

A03: `mintFrom` does not check or decrease the minting limit

[Severity: Low | Difficulty: Low | Category: Input Validation]

The `mint` function in the Alchemist calls `_checkMintingLimit` and `_mintingLimiter.decrease`, but the `mintFrom` function does not. This means that minting using an allowance doesn't count towards the minting limit.

Scenario

A user can entirely circumvent the minting limits by calling `approveMint` to give themselves an allowance, then using `mintFrom` to mint from their own CDP.

Recommendation

Add `_checkMintingLimit` and `_mintingLimiter.decrease` to `mintFrom` as well.

Status

Addressed in PR #20.

A04: Harvest can cause `balance == 0`

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

If a sufficiently-high number of shares are burned from the Alchemist (for example via withdrawals) after the price of a yield token has increased but before a harvest, the `expectedValue` of the token can become `0`. This will cause the next harvest to harvest the entire balance of the token in the Alchemist.

Once this happens, users become unable to deposit because `_issueSharesForAmount` causes a division by 0 when calculating the number of shares if `totalShares > 0` but `balance == 0`. Furthermore, because `expectedValue == 0` as well, withdrawals and liquidations are also blocked.

Scenario

Suppose that the price of a yield token is `p` when a harvest happens. Then, immediately after the harvest, `expectedValue == balance * p`. Then, suppose the following sequence of events happens:

1. The price of the token doubles to `2 * p`.
2. Users collectively withdraw half of the token's `totalShares` in the Alchemist. Both `totalShares` and `balance` of the token are halved, and `expectedValue` decreases by `(balance / 2) * (2 * p) == balance * p == expectedValue`. Therefore, `expectedValue == 0`.
3. Another harvest is triggered by the keepers, and since `expectedValue == 0` then `harvestable` is calculated to equal `balance`. Therefore, after the harvest `balance == 0`.

Recommendation

Add a pre-harvest step before operations that depend on the balance, calculating how much yield has been earned and transferring the corresponding amount of yield tokens from the balance to a harvest buffer. This guarantees that `expectedValue >= balance * p` before funds are withdrawn, and therefore `expectedValue` cannot go to `0` without `balance` going to `0`.

Status

Addressed in PR #49.

A05: `donate` does not call `lowerHasMinted`

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The `burn` function in the Alchemist calls `lowerHasMinted` after burning debt tokens to decrease the count of tokens minted, but the `donate` function does not.

Scenario

Since donating doesn't lower the total minted amount, if the number of debt tokens donated reaches the mint ceiling of the Alchemist, no tokens can be minted anymore unless the mint ceiling is increased.

Recommendation

Add `AlchemicTokenV2(debtToken).lowerHasMinted(amount)` to `donate` as well.

Status

Addressed in PR #43.

A06: `refreshStrategies` iterates over the wrong bounds

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The function for refreshing strategies in the TransmuterBuffer uses the list of underlying tokens in the Alchemist to iterate over all registered assets in the TransmuterBuffer, in the first for-loop below.

```
    /// @inheritdoc ITransmuterBuffer
   function refreshStrategies() public override {
     address[] memory supportedYieldTokens =
alchemist.getSupportedYieldTokens();
     address[] memory supportedUnderlyingTokens =
alchemist.getSupportedUnderlyingTokens();
     // clear current strats
     for (uint256 j = 0; j < supportedUnderlyingTokens.length; j++) {
       delete yieldTokens[registeredUnderlyings[j]];
     }

     uint256 numYTokens = supportedYieldTokens.length;
     for (uint256 i = 0; i < numYTokens; i++) {
       address yToken = supportedYieldTokens[i];

       (,address underlying,,,,,,,) =
alchemist.getYieldTokenParameters(yToken);

       yieldTokens[IDetailedERC20(underlying)].push(yToken);
     }
     emit RefreshStrategies();
   }
```

The above code only works as intended (deleting all strategies in the TransmuterBuffer) if it iterates over the registered assets in the TransmuterBuffer. Using `supportedUnderlyingTokens.length` defeats this, unless the number of underlying tokens in the TransmuterBuffer and in the Alchemist are the same.

Scenario

The Alchemist and TransmuterBuffer are in sync, keeping mappings of the same yield tokens and underlying tokens. If the Alchemist adds another underlying token, `refreshStrategies` will fail with an out-of-bounds error, until the TransmuterBuffer registers another underlying token.

Recommendation

Use `supportedUnderlyingTokens.length` instead of `supportedUnderlyingTokens.length` to iterate over all strategies.

Or add `require(supportedUnderlyingTokens.length == registeredUnderlyings.length)` for an added check that the contracts are in sync.

Status

Addressed in PR #54

A07: registerAsset and deRegisterAsset can drift out of sync with Alchemist

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

The TransmuterBuffer maintains a list of registered underlying tokens. It is intended to stay in sync with the Alchemist, but this is not enforced by the code. It is possible for the governance of the TransmuterBuffer to add arbitrary tokens, with unintended consequences.

Recommendation

`registerAsset` should check if the asset is supported by the Alchemist.

Status

Addressed in PR #54

A08: Price drop events can block claims from the Transmuter

[Severity: Medium | Difficulty: High | Category: Functional Correctness]

A Transmuter holds its transmuted tokens as yield tokens in its Alchemist via a CDP owned by the TransmuterBuffer. This means that, although the accounting logic of the Transmuter denotes users' claims in underlying tokens, the Transmuter is actually sensitive to price movements of the yield tokens.

If a price drop event happens that causes the total value of the TransmuterBuffer's CDP to fall below the current claimable amount in all its Transmuters, then the Transmuter cannot honor all its claims.

This is partially mitigated by a spread of risk, as the TransmuterBuffer can balance its portfolio of yield tokens so that it's unlikely that even a severe event in one of the tokens would cause it to become insolvent. Black swan events which bring down the price of many of the yield tokens at once are a bigger threat, but also far more unlikely.

If there is a severe loss in one of the yield tokens, and that loss is persistent, then the accounting logic of the Transmuters will be out of sync until the yield token has caught up. To mitigate this, governance should always immediately pause the Transmuter whenever one of its yield tokens has a price drop that needs to be `snapped` in place. The Transmuter should not be unpaused until all its transmuted tokens have been claimed.

Scenario

A Transmuter between alFOO and FOO has 50% of its claimable tokens deposited as the yFOO yield token. The yFOO token suffers a 20% price drop.

Assume there is only one user of the Transmuter (though the situation repeats when there are many users, the only difference being that in that case only those who claim late are affected). This user have X tokens claimable. If they try to claim all the X tokens, their transaction will fail. The maximum they can actually claim is X * (50% + 50% * 80%) = X * 0.9. At the same time, they cannot withdraw the missing 10% of claimable tokens as synthetic tokens, because those tokens are considered exchanged, and cannot be withdrawn.

The Transmuter does not have any builtin mechanism to adjust this issue. It would be contingent on governance to donate the missing amounts to the TransmuterBuffer to enable withdrawals.

Recommendation

One way to mitigate the issue is to allow governance to update the ratios of exchanged and unexchanged balances. A fall in yield token price would then require governance to update the exchanged and unexchanged ratios of the accounts, so that any unexchanged balance is increased, and the exchanged balance decreased with the same amount.

Status

Acknowledged by client. The Transmuter will be upgradeable. The risk of this event is deemed low. There are plans to mitigate this potential event in future upgrades. Both we and the client have suggested immediate mitigations[10] in the case such a price drop event takes place before a planned upgrade can be deployed.

[10] The simplest example would be a "haircut" constant inserted on all exchanged balances and moved to unexchanged balances, effectively moving some of every user's transmuted tokens back to being untransmuted.

A09: Short-term whaling of the Alchemist

[Severity: Medium | Difficulty: High | Category: Security]

A whale (an account with a very large amount of underlying or yield tokens) can snag much of the gains between two harvests, without having contributed to the actual yield earned in that time.

The attack is considered somewhat infeasible due to the large amount of capital on hand required, and the often small gains between harvests. The risk is highest when a yield token makes an unexpectedly large gain between harvests[11].

This is an attack on the rightful returns of a user's CDP. The intended behavior of Alchemix is that having an open CDP should be equivalent to holding the yield tokens directly in terms of risks and rewards (not accounting for fees paid to Alchemix). With this attack, while no user actually becomes poorer, they have not made as much money on their yield tokens as they should have. Therefore, other risk mitigations such as keeping the total yield tokens held by the Alchemix system weighted according to some risk don't address this issue, because the Alchemix system only loses some fees it would otherwise have earned, whereas the great losers are the users themselves.

Scenario

If a yield token has increased in value enough to warrant this attack, then what a whale can do is frontrunning the harvest transactions in the following way:

1. Buy a large amount of the yield token at the new, higher price.
2. Deposit the yield token in Alchemix.
3. Wait for a harvest to happen.
4. Withdraw the yield tokens and sell them.

The whale now has earned a cut of the credit from the harvest proportional to their share of yield tokens, even though their yield tokens didn't confer any additional yield to the system.

Recommendation

One way to mitigate this would be for the harvest to only give yields to those users who were invested before the previous harvest. That way, only users who had a yield token deposited for

[11] For an example of this happening in the wild, see the Cream Finance attack, where an attacker managed to steal tokens from Cream by *doubling* the value of a yield token. This essentially made anyone holding that yield token a winner, at the expense of the Cream system.

the full duration between two harvests would take part in those gains. However, the complexity of such an implementation might outweigh its benefits.

Another option would be to block deposits if the harvest gets too “loaded”. If it’s detected that the next harvest would result in a total yield payout over a certain percentage, the Alchemist could refuse to accept deposits. Once a harvest has happened, deposits would be possible again. This would likely cause little disturbance to the operations, because keepers are expected to harvest immediately after a certain yield target has been reached, but it would prevent front-running the harvest. That’s because the moment that the price of the yield token has increased significantly, deposits would be blocked, even in the same transaction as the price boost happens.

Status

Acknowledged by client. The Alchemist will be upgradeable. The risk of this attack is considered very low due to the capital intensity of it and the alternative costs in allocating that capital this way even for a short time. It is possible to perform the attack on the existing v1 Alchemists, but the Alchemix team claims they have not seen any such attack yet. Therefore it will not be mitigated before launch.

This finding is still considered important enough to mitigate in the future, and a mitigation is planned for a future upgrade.

A10: deRegisterAsset in TransmuterBuffer always fails

[Severity: Medium | Difficulty: Low | Category: Functional Correctness]

The function `registerAsset` in the TransmuterBuffer takes an underlying token and adds it to the set of registered underlying tokens. The function `deRegisterAsset` is intended to remove it from this set, but the function always reverts.

Note that it uses a variable, `success`, to check whether the token was found in the set, so that it can return a helpful error otherwise. But the variable is never modified in the loop, and this is always false at the end of the loop.

```
    function deRegisterAsset(IDetailedERC20 underlyingToken) external
override onlyAdmin {
    bool success = false;
    // only add to the array if not already contained in it
    for (uint256 i = 0; i < registeredUnderlyings.length; i++) {
      if (registeredUnderlyings[i] == underlyingToken) {
        delete transmuter[underlyingToken];
        registeredUnderlyings[i] =
registeredUnderlyings[registeredUnderlyings.length - 1];
        registeredUnderlyings.pop();
        TokenUtils.safeApprove(address(underlyingToken),
address(alchemist), 0);
        emit DeRegisterAsset(address(underlyingToken));
      }
    }

    if (!success) {
      // #if !(DISABLE_CUSTOM_ERRORS)
      revert IllegalState("De-register unsuccessful");
      // #else
      require(false);
      // #endif
    }
  }
```

Recommendation

This issue in isolation can be solved by setting `success = true` in the `if` body in the loop body. We would also recommend adding unit tests for `deRegisterAsset`. A simple happy-path test would catch this issue.

Status

Addressed in PR #54. `deRegisterAsset` was removed altogether.

A11: Yield token attack may be incentivized to leverage through Alchemix to boost their takeaway

[Severity: High | Difficulty:High | Category: Security]

We assume here that an attacker has found an exploit of a yield token by transiently manipulating the yield token price upwards. If an attacker can buy yield tokens, manipulate price up transiently, and cash out before the price drops again, that's a successful attack on the yield token. This would likely be a flash loan attack during a single transaction. Such an attack could of course also go through Alchemix by minting debt tokens and liquidating at a higher price. The naive attack would not result in any extra takeaway for the attacker compared to just buying, manipulating and selling the yield token. The goal of this issue is to show that there are ways in which the attacker could also siphon off funds from Alchemix, making the attack more profitable and damaging Alchemix.

The attack relies on the `depositFunds` function of the TransmuterBuffer being unprivileged. `depositFunds` causes the Alchemist to buy into a predetermined set of yield tokens. It may be bad from a DeFi lego perspective to let any user suddenly cause a predictable cash injection to a set of yield tokens, and allowing it increases the attack surface.

The consequences of this attack are similar to those in A08. The reason that this issue is listed on its own is that any attacker which can pull off such an attack can earn even more by also exacerbating the problem by going through Alchemix.

Scenario

Here's an example sequence of an attack, assuming weakness in a yield token. We imagine an attacker can buy a yield token from the issuer, increase it's redeemable price by 10% by some mechanism, then immediately redeem it for underlying tokens, stealing 10% of their invested value from other token holders. Assume the attacker has control of X yield tokens, with a "real" total value (measured in underlying tokens) of P and a manipulated value of 1.1*P.

1. Instead of just cashing out after manipulating the price, deposit the yield tokens to Alchemix.
2. Take out a maximum loan of 0.55*P (assume 50% collateralization ratio), then liquidate.
3. The Alchemist now redeems X/2 of the yield token, sends X/2 of the yield token back, and sends underlying tokens to the TransmuterBuffer. Alchemix now has 0.55*P underlying tokens, and an outstanding debt of 0.55*P. Note that at this point, Alchemix users have not been damaged by the attack, because they have gotten some of the manipulated price gains to compensate for the extra large loan they gave out.
4. Now the attacker calls depositFunds on the TransmuterBuffer, causing the TransmuterBuffer to buy back into the manipulated yield token. Assume that the TransmuterBuffer deposits 30% of the funds back into the yield token. Now it holds

0.15*X of the yield token more than it did before the attack. What's more, those yield tokens are in the TransmuterBuffer, belonging to the system (and no individual CDP), and were intended to back Transmuter claims.

5. The attacker cashes out of the manipulated yield token and makes off with the underlying value of the token + their Alchemix loan. Alchemix holds 0.15*X more of the manipulated yield token, than it would have had it not been attacked.

Note that for any attack on the yield token, regardless of if it goes through Alchemix or not, Alchemix users who use the yield token in their CDP will still suffer the price drop, but that is at their own risk and not different from if they had been holding the token directly. As long as their CDP is worth more than their loan, they are incentivized to repay or liquidate their position. Only if the price drop would bring a user's CDP collateral below their loan amount would the Alchemix system be left holding the bag. It is part of sound governance of Alchemix to set sensible collateralization parameters.

The above attack, however, leaves the Alchemix system holding an extra bag without a user accepting the risk.

Recommendation

Make `depositFunds` privileged to only keepers.

Status

Addressed in PR #22.

A12: Minting before a harvest can leave a CDP undercollateralized

[Severity: Medium | Difficulty: Medium | Category: Functional Correctness]

If a user mints from a CDP where the yield tokens have appreciated but the yield has not yet been harvested, the CDP might become undercollateralized after the next harvest, even though it was not undercollateralized immediately after minting (where the check for under-collateralization happens).

Scenario

Suppose the minimum collateralization ratio is 2.

1. User deposits `x` yield tokens when the price is `p`.
2. Price doubles to `2 * p`. The user's collateral is now worth `2 * p * x`.
3. User mints `p * x` debt tokens, half of the current collateral (obeying the minimum collateralization ratio).
4. The keepers trigger a harvest. Half of the yield tokens are harvested, so the user's collateral is `p * x`. The user's collateralization ratio is now 1.
5. Even after having their CDP updated by `poke`, the user's collateralization ratio might not immediately recover, since other users may have been issued new shares and will get some of the yield from the first user's tokens.

The collateralization ratio might even drop below 1, for example if in step 2 the price more than doubled, and the user still minted half of the collateral in step 3.

Recommendation

Related to A04, users can be prevented from minting more than they would be able to mint after harvest by pre-harvesting all the yield tokens in the user’s CDP before minting. This ensures that the test for undercollateralization performed by `_validate` is accurate.

Status

Fix implemented in PR #49.

Informative findings

This section covers those findings that do not necessarily need to be addressed, but that we want to make users, keepers, owners and developers aware of. They are not security vulnerabilities. Examples of categories of findings in this section are security trade-offs, important considerations for governance for maintaining security, code style, gas savings and gotchas.

B01: ABIEncoderV2 no longer experimental

[Severity: Low | Difficulty: N/A | Category: Best Practice]

Because the ABI coder v2 is not considered experimental anymore, it can be selected via "`pragma abicoder v2`" since Solidity 0.7.4.[12] In Solidity 0.8.0 and above, it's enabled by default, and the pragma therefore has no effect.

The "`pragma experimental ABIEncoderV2`" is used in the following locations contracts, all using compiler version ^0.8.9:

- contracts/AlchemicTokenV2.sol
- contracts/test/yearn/YearnVaultMock.sol
- contracts/test/ERC20Mock.sol
- contracts/interfaces/yearn/VaultAPI.sol
- contracts/interfaces/aave/ILendingPool.sol

Recommendation

Remove the unnecessary pragmas.

Status

Addressed in PR #69.

[12] https://docs.soliditylang.org/en/v0.8.9/layout-of-source-files.html#abiencoderv2

B02: `SafeCast.toUint256` reverts when input is `0`

[Severity: Low | Difficulty: N/A | Category: Best Practice]

The function `toUint256` in the `SafeCast` library currently requires the input to be strictly greater than `0`, rather than greater than or equal.

At the moment this does not affect any behaviors, as the only place where the function is called is in `TransmuterBuffer.getCredit`, and always with a strictly positive input. However, if a future upgrade causes this function to be called without such a guard, while expecting it to work for an input of `0`, it could cause the function to unexpectedly revert and prevent an operation from being completed.

Recommendation

Replace `y > 0` with `y >= 0` in the implementation of `toUint256`.

Status

Addressed in PR #23.

B03: Failing `assert` in `_issueSharesForAmount`

[Severity: Low | Difficulty: N/A | Category: Best Practice]

The `assert(shares > 0)` in the function `_issueSharesForAmount` of the `AlchemistV2` contract can be violated if the `amount` parameter is `0` (in which case `shares` will also be calculated to be `0` by the `_convertYieldTokensToShares` function). Since `_issueSharesForAmount` is called by the external functions `deposit` and `depositUnderlying` and the `amount` argument is provided directly by the user, nothing prevents this value from being `0`.

According to the Solidity documentation, as of version 0.8.0, Panic exceptions like those generated by `assert` no longer consume all remaining gas. Even so, the use of `assert` for a condition that can be violated during execution is still not recommended, since it can be misleading and will be flagged as a bug by code analysis tools.

Recommendation

Replace `assert(shares > 0)` by either `assert(amount == 0 || shares > 0)` or `require(amount > 0)`. In the latter case the test can be moved to the top of the function or even to the external functions.

If the function is modified so that it does not revert in the case that `amount == 0`, the `if` statement underneath the assert should also be updated to only insert the token into the `depositedTokens` set if `shares > 0`.

Status

Addressed in PR #24.

B04: Redundant bounding in `Limiters.update()`

[Severity: Low | Difficulty: N/A | Category: Best Practice]

When assigning `self.lastValue`, the `update()` function in `Limiters.sol` bounds the return value of the `get()` function by `self.maximum`. This is unnecessary since this value is already bounded inside the `get()` function.

Recommendation

In `update()`, replace `self.lastValue = value > self.maximum ? self.maximum : value` with `self.lastValue = value`.

Status

Addressed in PR #25.

B05: Governance must manually ensure TransmuterBuffer invariants

[Severity: Low | Difficulty: N/A | Category: Input Validation]

The TransmuterBuffers deposit funds back into the Alchemist according to a list of weights and a list of yield tokens. The weights are set by the admin role.

If any of the yield tokens in the list of weights is not supported by the Alchemist, then TransmuterBuffer will not be able to deposit any tokens until its weights are updated.

A related issue is if governance adds the wrong yield token to a weighting, but it is one which is actually supported by the Alchemist, only for a different underlying token. In such a case the safeguards in *depositFunds* fail. However, it only means that the deposits don't use up the expected number of the correct underlying token.

Scenario

The TransmuterBuffer has assigned some weight to yield token X, which was previously supported by the Alchemist. The Alchemist has since stopped supporting the yield token. Now keeper transactions to deposit funds into the Alchemist will fail, because one of the deposit transactions to the Alchemist will fail.

Recommendation

It would be possible to check that each token is supported as its weights are added, but this does not fully secure against the above scenario. The Alchemist could notify the buffer of changes, but this would increase complexity. We advise that some operational process is put in place to guarantee that whenever governance updates the Alchemist or the TransmuterBuffer, that it is verified that all tokens in the weightings in the TransmuterBuffer are supported by the Alchemist. If governance is passed to a DAO, which may approve some transactions but not others, this logic could also be implemented in a smart contract for higher reliability.

Status

Acknowledged by client.

B06: Unnecessary indirection in looking up TransmuterBuffer weights

[Severity: N/A | Difficulty: N/A | Category: Best Practice]

The weightings mapping takes strings as keys, each string specifying a “weight type”. The weight type is either “burn”, or a key that should be tied to a specific underlying token.

This means that two mappings must be manually kept in sync: the one containing weights, and the mapping from underlying token to a weight type. It also means that care must be taken when updating either, as between transactions underlying token may have no weights assigned.

Recommendation

Use the underlying token as key directly to the mapping of weightings. The debt token can be used as the key for burning.

Status

Addressed in PR #53.

B07: Loss of a yield token calculated as a fraction of current value rather than expected value

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

In the `_loss` function of the `AlchemistV2` contract, the loss is calculated as

```
(expectedValue - currentValue) * 1e4 / currentValue
```

This calculates the loss as a fraction of `currentValue` instead of `expectedValue`. This means, for example, that if `currentValue` is less than half of `expectedValue`, the calculated loss exceeds 100%.

Recommendation

Replace `currentValue` by `expectedValue` in the denominator.

Status

Addressed in PR #29.

B08: TransmuterBuffer uses a mix of access control systems

[Severity: Low | Difficulty: N/A | Category: Best Practice]

The TransmuterBuffer makes use of a mapping of `keepers`, as well as using the AccessControl library from OpenZeppelin. Instead, it could use the AccessControl library for all access control.

Recommendation

Set up a "`KEEPERS`" role in TransmuterBuffer, and set and revoke that role instead of modifying the `keepers` mapping.

Status

Addressed in PR #90.

B09: Transmuter calls AccessControl functions unconventionally

[Severity: Medium | Difficulty: N/A | Category: Best Practice]

The AccessControl library implements roles, and ways to grant and revoke permissions. However, the TransmuterBuffer has a function called `setSentinel` which looks as follows:

```
function setSentinel(address _newSentinel) external {
  onlyAdmin();
  _setupRole(SENTINEL, _newSentinel);
}
```

It calls `_setupRole`, which isn't intended to be called directly apart from in the constructor. Instead, `grantRole` should be used. `grantRole` is also inherited from the AccessControl contratract, making the above function superfluous.

Recommendation

Remove the above function, and verify in the unit tests that it can be replaced by calls to `grantRole`.

Status

Addressed in PR #119.

B10: Yield tokens and underlying tokens can become unsupported

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

Yield tokens can become unsupported, even if there are CDPs that contain them. Underlying tokens can also become unsupported, in which case all their yield tokens become unsupported.

The consequence for users with open CDPs is that they can no longer repay their loans using these yield tokens or underlying tokens. What's more, they can no longer liquidate them. All they can do is burn synthetic tokens (to repay), and then withdraw the yield tokens

Scenario

User Alice has a CDP backed by yFOO, and has maxed out an alFOO loan from her CDP. A while later, before the debt has been paid off by the yield, the yFOO token becomes unsupported due to a governance decision which updates the FOO alchemist. Alice had bought her yFOO using leverage on another platform, and price movements are causing her to risk liquidation, unless she can liquidate her Alchemix position and pay back some of the loan. She doesn't currently have the liquidity to pay off the loan.

Fortunately for Alice, there is a workaround, if she is savvy or there is a service which can help her: she can flash-mint alFOO and in the same transaction repay her loan, withdraw her yFOO and sell enough of it to pay back her flash-mint debt.

Recommendation

There is no recommended action for the protocol. It's important that users are aware that the `liquidate` function may not always be available for all the yield tokens in their CDPs, and plan accordingly.

B11: General considerations about price drop events

[Severity: Low | Difficulty: N/A | Category: Risk Profile]

In the case of a price drop in a yield token, CDPs in the Alchemist might become undercollateralized until the price recovers. If this happens, withdrawing and minting become blocked by the `_validate` function until the CDP recovers (either via an increase in price, more collateral being deposited or debt being repaid). Independently of the collateralization ratio, the `_checkLoss` function also prevents deposits, withdrawal of underlying tokens and liquidation if the yield token suffers a price drop above the maximum loss configured in the protocol. This is meant to prevent users from realizing a sudden transient loss before the price recovers. Functions that unwrap underlying tokens from the yield tokens, such as `withdrawUnderlying` and `liquidate`, also may require the user to provide the maximum acceptable loss as a parameter (if the yield tokens supports such slippage protection), so that users that have not noticed the price drop event will not accidentally realize losses.

If a yield token doesn't recover from a loss, governance has the option of snapping the expected value of the token to its current value. This realizes the loss, but recovers the deposit, withdrawal and liquidation functionalities.

In the Alchemist, price drop events which are transient in nature or due to a temporary accounting bug (something which has occurred previously in Yearn tokens) would mean that users are incentivized to wait for the price to be restored before withdrawing underlying tokens, to avoid realizing the transient losses. However, there is no such incentive for users of the Transmuter. Recall that the claimable tokens in the Transmuter are stored in a CDP owned by the TransmuterBuffer. The users of the Transmuter have no incentive to wait for price drop events to pass, but can claim their transmuted tokens immediately, selling off a disproportionate amount of yield tokens from the TransmuterBuffer CDP in the process. See A08: Price drop events can block claims from the Transmuter. However, they can only do so if the price drop is less than the `maximumLoss` for the relevant yield tokens. The variable `maximumLoss` can be any value between 0 and 100 %. Thus when setting this variable, governance should take into account the potential for Transmuter users to either withdraw at the dropped price (if the price drop is below the `maximumLoss`) or become stuck and unable to withdraw until the price is `snapped`.

B12: A malicious keeper can steal one entire harvest

[Severity: Medium | Difficulty: Low | Category: Security]

First of all note that keepers are highly trusted entities. They are intended to be either immutable smart contracts, upgradeable smart contracts with trusted governance, or EOAs or multisigs with trust assumptions close to that of the governance addresses.

There is a known attack that whales can perform on harvests, see A09: Short-term whaling of the Alchemist.

That attack is mostly considered infeasible, though it is something an MEV searcher or other kind of value extractor could be on the lookout for, treating it like an arbitrage.

However, if a keeper account is compromised or acts maliciously, they can very easily steal (almost) an entire harvest, because unlike a whale, they do not need to have cash on hand and wait for a harvest to occur before they earn credit. Instead, a keeper can flash loan any amount of tokens it needs, make a deposit into the Alchemist that dwarfs all the other deposits, trigger a harvest, and then withdraw their deposit and repay their flash loan, all in one transaction.

Scenario

A keeper can perform the following attack on a yield token:

1. Take out a large flash loan.
2. Buy a large amount of the yield token.
3. Deposit the yield token in Alchemix.
4. Trigger a harvest on the yield token.
5. Withdraw the yield tokens and sell them.
6. Repay the flash loan.

The keeper has now earned approximately the full value of the harvest, given that the flash loan was large enough. If `maximumExpectedValue` of a token is M, current `expectedValue` value of all tokens in CDPs ic C, then the attacker should be able to steal (roughly) $\frac{M-C}{M}$ of all the yield of that yield token (since last harvest).

Given that such an attack can be detected, and governance can revoke the keeper’s privileges, it is likely in the keeper’s interest to perform this attack on all the yield tokens in all the Alchemists on which it is privileged.

Recommendation

The recommendations in A09: Short-term whaling of the Alchemist apply here as well.

Another option is to disallow keepers to have CDPs, and CDP owners to become keepers, through simple `requires` clauses.

Status

Acknowledged by client. Keepers are considered trusted enough. Any whitelisted keeper is assumed to be either a) a trusted EOA or b) a trusted contract called directly by a trusted EOA.

B13: The maximum allowed value and maximum allowed loss of a yield token is not configurable

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

The parameter `maximumExpectedValue` of a yield token in the Alchemist controls how much balance (as measured in underlying tokens) the Alchemist is allowed to hold. Once that limit is reached, the yield token can still be harvested, withdrawn and liquidated, but not deposited.

This is a useful measure to control the maximum total value locked of different assets. However, there is no function for configuring it, and when the ceiling is reached, it cannot be increased, ever.

Recommendation

Create a governance-only function for increasing the maximum expected value of a yield token, and the maximum acceptable loss of a yield again.

Status

Addressed (in multiple PRs).

B14: Documentation error in LiquidityMath library

[Severity: Low | Difficulty: N/A | Category: Best Practice]

The `calculateProportion` function reports performing the wrong operation, claiming to divide its inputs rather than multiplying them.

```
    /// @dev Calculate a uint256 representation of x / y using
FixedPointMath
  ///
  /// @param  x The numerator
  /// @param  y The denominator
  /// @return z The resulting proportion
  function calculateProportion(uint256 x, FixedPointMath.Number memory y)
internal pure returns (uint256 z) {
    z = y.mul(x).truncate();
  }
```

The intended use of `calculateProportion` is to take as its second input a proportion, i.e. a fractional number such as 0.02 or 1.24, and to scale the first input accordingly.

Recommendation

Update the documentation.

Status

Addressed in PR #126.

B15: `totalMinted` does not decrease if AlchemicToken is burned without going through the Alchemist

[Severity: Medium | Difficulty: N/A | Category: Functional Correctness]

The `totalMinted` mapping is used by the `AlchemicTokenV2` contract to keep track of how many tokens have been minted by an Alchemist. Once this total reaches the Alchemist's mint ceiling, it cannot mint further tokens until some tokens have been burned.

However, although `totalMinted` increases automatically when tokens are minted, it does not decrease automatically when tokens are burned, requiring calling `lowerHasMinted` afterwards to do so. The Alchemist does this consistently (after the fix to A05), but other contracts and EOAs are also able to burn alTokens. The `lowerHasMinted` function is marked as `onlyWhitelisted`, so it should not be callable by these accounts. Therefore, if alTokens are burned without going through the Alchemist, they do not count towards decreasing the `totalMinted` amount.

Scenario

If a user burns *n* alTokens without going through the Alchemist (by calling either `burn` or `burnFrom` directly in the AlchemicToken), the `totalMinted` amount will never drop below *n* again without intervention by the governance. If the total number of tokens burned in this way over time reaches the mint ceiling, the Alchemist will be unable to mint further tokens unless the ceiling is increased.

Recommendation

Unlike A05, this scenario should not happen as part of the normal operation of the protocol. alTokens should only be burned by the Alchemist, and users should have no incentive to burn tokens themselves, as they are effectively spending assets they control in exchange for nothing.

However, if this does happen for any reason, the discrepancy in the `totalMinted` amount cannot be corrected without intervention by governance. One way that this could be done would require the admin to temporarily whitelist themselves in order to call `lowerHasMinted`. This would also require finding the correct amount to correct `totalMinted` by, for example by using `totalSupply()` to find the number of tokens in circulation.

If the number of tokens burned in this way reaches the mint ceiling for the Alchemist, the governance can also mitigate the problem by increasing the ceiling, although `totalMinted` will still be inaccurate.

Status

Acknowledged by client. This is an artifact of maintaining backwards-compatibility with the existing alTokens. Since burning tokens directly is never in the interest of a user, and there is no real harm to the protocol from them doing so, this implementation is left as is.

B16: Debt token converted to `AlchemicTokenV2` can be `AlToken`

[Severity: Low | Difficulty: N/A | Category: Best Practices]

The Alchemist converts the debt token to the `AlchemicTokenV2` contract type to call `lowerHasMinted`, but some of the debt tokens will use the original `AlToken` contract rather than the new `AlchemicTokenV2` contract.

Recommendation

Since `AlToken` has a `lowerHasMinted` function with the same signature as the one in `AlchemicTokenV2`, this should not cause any issues for the operation of the Alchemist.

However, it might be desirable to have the Alchemist call `lowerHasMinted` through an interface instead (there is already an `IAlchemicToken` interface, although it does not currently define `lowerHasMinted` and is not inherited by `AlchemicTokenV2`). This approach would have the following benefits:

- It clarifies the intent that the debt token may not necessarily be an instance of the `AlchemicTokenV2` contract.
- The `AlchemicTokenV2` contract can be made to inherit the interface, guaranteeing that it is compatible.
- Although the currently-deployed `AlToken` contract does not inherit the interface, a version of the `AlToken` contract that does inherit the interface can be deployed locally to test for compatibility.
- If an Alchemist is deployed in the future with a debt token that uses a new version of the AlchemicToken contract, the new contract can also inherit the interface in order to ensure compatibility.

Status

Addressed in PR #59.

B17: Remaining TODOs and FIXMEs in documentation

[Severity: Low | Difficulty: N/A | Category: Best Practice]

There are remaining TODOs and FIXMEs in the comments. There are FIXMEs in the TokenUtils library indicating that better error messages should be implemented. The remaining TODOs are in the interfaces IERC20Minimal and IERC20Meta.

Status

Addressed.

Checked properties

There is more to an audit than finding bugs. We also want to report what errors we have ruled out, and what properties we have verified.

C01: Compiler version

^0.8.9 is used in all built and deployed contracts, and all other contracts use either ^0.8.x or >=0.x.y

The latest Solidity release[13] fixes no bugs which are relevant to this release.

Update: The codebase has been updated since the reviewed commit. ^0.8.11 is used in all built and deployed contracts, and all contracts that do not use ^0.8.11 use >=0.5.0.

[13] https://github.com/ethereum/solidity/releases/tag/v0.8.10

C02: Conversions between quantities

Numerical amounts manipulated by the `AlchemistV2` contract generally represent one of five quantities: underlying tokens (U), yield tokens (Y), shares (S), normalized underlying tokens (N), or weight (W). Each of these quantities uses a different number of decimal places:

- Underlying tokens have a number of decimals that varies depending on the type of token, and is stored in `_underlyingTokens[underlyingToken].decimals`.
- Yield tokens also have a number of decimals that varies depending on the type of token, stored in `_yieldTokens[yieldToken].decimals`.
- Shares of a yield token use the same number of decimals as the yield token.
- Normalized underlying tokens are normalized to use the same number of decimals as the Alchemist's debt token.
- Weight for a given yield token uses a number of decimals equal to `(18 + debtTokenDecimals - yieldTokenDecimals)`.

The following formulas are used to convert between these amounts in the Alchemist contract, and we have manually inspected them to make sure that they result in the correct number of decimals:

```
S = Y * totalShares / balance
```

`_yieldTokens[yieldToken].totalShares` (S) and `_yieldTokens[yieldToken].balance` (Y) keep track respectively of the total balance of shares and tokens of a given `yieldToken` in the Alchemist. Initially, both are the same, making the conversion between shares and yield tokens one-to-one. Both increase when users deposit collateral and decrease when users withdraw or liquidate collateral. However, when yield is harvested, an amount of yield tokens corresponding to the accrued yield is converted to underlying tokens and sent to the Transmuter. This amount is subtracted from `balance`, while `totalShares` remains the same. Keeping track of `totalShares` separately from `balance` is important since the percentage of shares a user has in the Alchemist determines the percentage of yield that they receive at harvest.

```
U = Y * price / 10 ** decimals
```

The `price` (U) of a yield token in underlying tokens can be obtained from the `price()` function of the corresponding token adapter. `_yieldTokens[yieldToken].decimals` is used to normalize the result to the number of decimals of the underlying token.

```
N = U * conversionFactor
```

`_underlyingTokens[underlyingToken].conversionFactor` (N / U) for a given `underlyingToken` is equal to `10 ** (debtTokenDecimals - underlyingTokenDecimals)`. Underlying tokens with more decimals than the debt token are not accepted by the Alchemist.

```
W = N * 1e18 / shares
```

Weight represents credit accrued per share, either from harvests or donations. The value of `shares` (S) in the denominator will vary depending on how many shares the credit is being distributed amongst: for harvests it is equal to `totalShares`, while for donations it is equal to `totalShares` minus the number of shares of the donating account.

Additionally, the Alchemist also manipulates dimensionless values representing different ratios, each also using a specific number of decimals:

- CDP collateralization ratios use 18 decimals.
- The loss of a yield token is calculated with 4 decimals.
- The percentage of the harvested yields taken as a protocol fee is stored with 4 decimals.

We have created a version of the `AlchemistV2.sol` contract where variables are annotated with the type of quantity they represent using the notation `/*$ ... $*/` after the type. This version can be found in the following link:
https://github.com/alchemix-finance/alchemix-v2-auditable/blob/annotations/contracts/AlchemistV2.sol

Using this annotated version of the contract, we checked that conversions between these quantities are being performed correctly, and the quantities being passed to functions are compatible with what the function expects. We followed the following methodology:

1. We annotated all state variables, local variables and function parameters with the type of quantity they are expected to contain according to documentation and how they are used in the code.
2. We checked that every function uses variables internally in a way that is consistent with their annotation.
3. We checked that all conversions between different quantities follow the conversion formulas.
4. For every state variable, we checked that every use of that variable inside a function is consistent with its annotation.
5. For every function, we checked that every call of that function in the code receives arguments that are consistent with the annotation of the function's parameters.

C03: Critical section on `liquidate`

The `liquidate` function in `AlchemistV2.sol` includes a critical section within which the only external calls that can occur are read-only functions. We have verified that indeed no external calls that can modify state occur inside the critical section. The only external call that is performed is to the `view` function `ITokenAdapter.price()`, called from `_convertYieldTokensToUnderlying` (itself called from `_sync`).

C04: Mutex

We have checked that the `Mutex` contract is correctly implemented and that all external functions in the `AlchemistV2` contract that make external calls use the `lock` modifier to protect against reentrancy. Additionally, since the modifier is only used on `external` functions, there is no risk of deadlock where one locked function needs to call another.

C05: Alchemist Invariants

Invariant A1

Invariant A1: Assume all CDPs are fully updated (using _poke) and no rounding errors. Let m be the amount of debt tokens minted by the Alchemist, b the amount of debt tokens burned by the Alchemist, d the sum of all debts in the Alchemist, and t the amount of underlying tokens sent to the TransmuterBuffer from the Alchemist. Then, $m = b + d + t$. Note that if a CDP has credit (negative debt) this amount is *subtracted* from d.

Initially, $m = b = d = t = 0$. The only functions that update m are `mint(amount, recipient)` and `mintFrom(owner, amount, recipient)`, both of which increase it by `amount` by calling `TokenUtils.safeMint(debtToken, recipient, amount)`. Both also increase d by the same amount by calling `_updateDebt(owner, SafeCast.toInt256(amount), _sadd)`. The only other operation called by these functions that can affect the relevant values is `_poke(owner)`, but since we assume that all CDPs are fully updated, `_poke` returns without changing any state. Therefore, whenever m is updated the invariant is maintained.

The only functions that update b are `burn(amount, recipient)` and `donate(yieldToken, amount)`, both of which increase it by `amount` by calling `TokenUtils.safeBurnFrom(debtToken, msg.sender, amount)`.

In the case of `burn`, other than `_poke` the only relevant operation called is `_updateDebt(recipient, SafeCast.toInt256(amount), _ssub)`, which decreases d by `amount`. Therefore $b + d + t$ is kept constant, maintaining the invariant.

In the case of `donate`, the accrued weight w of `yieldToken` is increased to $w + 10^{18}a / (s - s')$, where a is `amount`, s is the total number of shares of `yieldToken` in the Alchemist and s' is the number of shares of `yieldToken` owned by `msg.sender`. It also updates `msg.sender`'s last accrued weight to this value. If all CDPs are then updated using `_poke`, this also translates to decreasing d by `amount`:

- For the `msg.sender` of `donate`, `lastAccruedWeight` is equal to the current `accruedWeight` of `yieldToken`. Therefore, following the implementation of `_poke`, their debt will remain the same.
- For every other CDP p, since they were updated before calling `donate`, their `lastAccruedWeight` is w. Therefore, following the implementation of `_poke`, the debt of p will decrease by $(10^{18}a / (s - s'))s_p / 10^{18} = a\, s_p / (s - s')$, where s_p is the number of shares of `yieldToken` in p.

Therefore, the total debt in the Alchemist decreases by $\sum_p (a\, s_p / (s - s')) - a\, s' / (s - s') = a (\sum_p s_p) / (s - s') - a\, s' / (s - s') = a (\sum_p s_p - s') / (s - s')$. Since $\sum_p s_p = s$ (by Invariant A2), this is equal to a (s

$- s') / (s - s') = a$. Therefore, the total debt d decreases by `amount`, and therefore $b + d + t$ remains the same, preserving the invariant.

There are three functions that send underlying tokens to the TransmuterBuffer: `repay(underlyingToken, amount, recipient)`, `liquidate(yieldToken, shares, params)`, and `harvest(yieldToken, params)`. Both `repay` and `liquidate` send an amount a of underlying tokens to the TransmuterBuffer while also calling `_updateDebt` with the same amount a, only normalized to the number of decimals used by the debt token. In the case of `harvest`, the amount a sent to the TransmuterBuffer is also normalized and passed to the `_distribute` function, which increases the accrued weight w of `yieldToken` to $w + 10^{18}a / s$, where s is again the total number of shares of `yieldToken` in the Alchemist. If all CDPs are then updated using `_poke`, the total debt d will decrease by $\sum_p (a\, s_p / s) = a\, (\sum_p s_p) / s = a\, s / s = a$. Therefore, in each case t increases and d decreases by a, preserving the invariant.

The total debt d is only updated in `_updateDebt` and `_poke`. All functions that call `_updateDebt` have been addressed above. Meanwhile, `_poke(owner, yieldToken)` returns without making any changes unless the last accrued weight of `yieldToken` for `owner` is different from the current accrued weight of `yieldToken`. Since the last accrued weight is set to the current accrued weight after `_poke` updates the CDP, this can only happen if the accrued weight changed since the last time the CDP was updated by `_poke`. This can only happen in the functions `donate` and `_distribute`, the latter only called by `harvest`. Both of these cases have been addressed above.

Therefore, whenever one of m, b, d or t changes, another changes as well in such a way that the invariant is preserved.

Invariant A2

Invariant A2: The total number of shares of a yield token is equal to the sum of the shares of that yield token over all CDPs.

The total number of shares of `yieldToken` is given by `_yieldTokens[yieldToken].totalShares`, while the shares of `yieldToken` in a CDP is given by `_accounts[owner].balances[yieldToken]`. Both of these start at 0, the first being initialized in `addYieldToken` and the second by the initial value of `_account`. There are only two functions that update these variables, namely `_issueSharesForAmount` and `_burnShares`. Both of them update the total number of shares and the number of shares of exactly one CDP by the same value.

Invariant A3

Invariant A3: Let b be the balance and t the total number of shares of a given yield token. Then, $b \leq t$, and $b = 0$ if and only if $t = 0$.

The balance of `yieldToken` is stored in the variable `_yieldTokens[yieldToken].balance`, while the total number of shares is stored in `_yieldTokens[yieldToken].totalShares`. Initially, both are equal to 0. The total number of shares t can only be updated via the functions `_issueSharesForAmount` and `_burnShares`.

Every function that calls `_issueSharesForAmount` or `_burnShares` first calls `_preHarvestToken` or `_preHarvestAccount`, which may update b to $b - (b \cdot p - e) / p = b - b + e / p = e / p$, where e is `_yieldTokens[yieldToken].expectedValue` and p is the token's current price in underlying tokens. Since this update only happens if $b > e / p$ (therefore the value of b cannot increase), and since $e = 0$ only if $b = 0$ (by Invariant A7), the invariant is maintained after pre-harvest (we will consider the effect of `_preHarvestAccount` on other tokens at the end).

`_issueSharesForAmount` increases t by a if t is 0 and $(a \cdot t / b)$ otherwise, where a is the value passed to the function as the parameter `amount`. Every function that calls `_issueSharesForAmount` also calls `_sync`, passing the same value a as the parameter `delta`, and `_uadd` as the `op` parameter. This causes the balance b to be increased by a. Assuming the invariant holds initially, if $t = 0$ then $b = 0$, and therefore both are updated to a, which preserves the invariant. If $t > 0$, then by the second part of the invariant $b > 0$ as well, and since $t \geq b$ then $(t / b) \geq 1$. This means that $(a \cdot t / b) \geq a$ (since a cannot be negative), and therefore $t + (a \cdot t / b) \geq b + a$, preserving the first part of the invariant. The second part is preserved since neither t nor b decreases in value.

`_burnShares` decreases t by the value s passed as the parameter `shares`. This function is called in the following functions: `withdraw`, `withdrawFrom`, `withdrawUnderlying`, `withdrawUnderlyingFrom`, and `liquidate`. All of these also call `_sync`, passing `_usub` as `op` and a value a as `delta` calculated by calling `_convertSharesToYieldTokens` on s. Following the implementation of `_convertSharesToYieldTokens`, a will be equal to s if $t = 0$, and $(s \cdot b / t)$ otherwise. The case where $t = 0$ doesn't need to be considered, since in that case subtracting s from t will cause a revert. Therefore, we only need to consider the case where t is updated to $t - s$ and b is updated to $b - (s \cdot b / t)$. We can also ignore the case where $t < s$, since it would likewise cause a revert. Assuming the invariant, we have that $t \geq b$, and therefore $t \cdot (t - s) \geq b \cdot (t - s)$ (since $t \geq s$). Dividing both sides by t, we get $t - s \geq b \cdot (t - s) / t = b \cdot (1 - s / t) = b - (s \cdot b / t)$. Therefore, the first part of the invariant is preserved. Furthermore, $t - s = 0$ iff $t = s$ iff $t \cdot b / t = s \cdot b / t$ iff $b = s \cdot b / t$ iff $b - s \cdot b / t = 0$. Therefore, the second part is also preserved.

The only remaining ways to update b but not t are via pre-harvest. This includes `_preHarvestToken` in `harvest` and `donate`, `_preHarvestAccount` in `mint` and `mintFrom`, as well as calls of `_preHarvestAccount` when withdrawing other tokens. Since, as mentioned before, pre-harvest either leaves b the same or updates it to e / p if $e / p < b$, and since $e = 0$ only if $b = 0$, the invariant is preserved.

Invariant A4

Invariant A4: Unless the token has suffered a loss, every operation that changes the balance or expected value of a yield token leaves the expected value equal to the current value, calculated by multiplying the price of the token by its balance.

Let b be the balance of the yield token, e its expected value, and p its current price. Then, the token's current value is calculated by $b \cdot p$. A loss on the yield token means that $b \cdot p < e$.

There are four functions that change the balance or expected value of a yield token: `_preHarvestToken`, `_preHarvestAccount`, `_sync` and the admin function `snap`. Of those, `snap` explicitly sets $e = b \cdot p$. Pre-harvest either does nothing if $b \cdot p \leq e$ or updates b to $b - (b \cdot p - e) / p = b - b + e / p = e / p$ otherwise. The latter case leaves $b \cdot p = e$, while the former case can only happen if the token has suffered a loss or if $b \cdot p = e$ already.

The last function to consider, `_sync`, is only ever called after pre-harvest. Therefore, we can assume that $b \cdot p = e$ when `_sync` is called. The function then updates the balance from b to $op(b, \Delta)$ and the expected value from e to $op(e, \Delta \cdot p)$, where Δ and op are function parameters, and op is either unsigned addition or subtraction. Since $e = b \cdot p$, then $op(e, \Delta \cdot p) = op(b \cdot p, \Delta \cdot p) = op(b, \Delta) \cdot p$ (since op is either addition or subtraction). Therefore, the relationship is preserved.

Invariant A5

Invariant A5: Assume the `AddressSet` data structure behaves correctly. Then, a user has balance in a token if and only if that token is in the set of deposited tokens for that user.

Initially, a user has a balance of 0 for all yield tokens, and the `depositedTokens` set is empty. The only functions that update a user's balance in a token or the `depositedToken` set are `_issueSharesForAmount` and `_burnShares`. The former only increases the balance, and adds the token to `depositedTokens` if the previous balance was 0 (note that currently the `assert` statement in `_issueSharesForAmount` prevents the function from completing execution if the amount added to the balance is 0; if the `assert` is removed as recommended in B03 then the code must ensure that the function either reverts or does not add the token to the set if the amount is 0). The latter only decreases the balance, and removes the token from `depositedTokens` if the balance after decreasing is 0. Therefore, both preserve the invariant.

Invariant A6

Invariant A6: Liquidating some amount of a debt will have the exact same result as repaying that amount of debt after withdrawing the same amount of collateral.

We will show that successfully calling `liquidate(yieldToken, shares, params)` has the same effect as successfully calling `withdrawUnderlying(yieldToken, shares, recipient, params); repay(underlyingToken, amountUnderlying, recipient)`, where `underlyingToken` is the underlying token of `yieldToken`, `amountUnderlying` is the amount of underlying tokens corresponding to `shares`, and `recipient` is the caller. We also use `amountYield` to represent the amount of yield tokens corresponding to `shares`.

There are three caveats that we disconsider for the sake of proving this property:

1. `liquidate` and `repay` affect different limiters: `liquidationLimiter` and `repayLimiter`, respectively. We ignore the effect on the limiters.
2. `liquidate` calls `_poke` only on `yieldToken`, while `repay` and `withdrawUnderlying` call it on the entire CDP. We ignore this since `poke` can always be called afterwards, and its effect is transparent from the point of view of the user.
3. Similarly, `liquidate` calls `_preHarvestToken` to pre-harvest only `yieldToken`, while `withdrawUnderlying` calls `_preHarvestAccount` to pre-harvest all tokens in the CDP. We ignore the effect on the other tokens, since they will be pre-harvested anyway before any future operation that depends on their balance[14].

A successful call of `liquidate(yieldToken, shares, params)` makes the following calls that affect state:

- `_preHarvestToken(yieldToken)`
- `_unwrap(yieldToken, amountYield, address(this), params)`
- `_poke(msg.sender, yieldToken)`
- `_burnShares(msg.sender, yieldToken, shares)`
- `_updateDebt(msg.sender, SafeCast.toInt256(_normalizeUnderlying(underlyingToken, amountUnderlying)), _ssub)`
- `_sync(yieldToken, amountYield, _usub)`
- `TokenUtils.safeTransfer(underlyingToken, transmuter, amountUnderlying)`
- `IERC20TokenReceiver(transmuter).onERC20Received(underlyingToken, amountUnderlying)`

A successful call of `withdrawUnderlying(yieldToken, shares, recipient)` makes the following calls that affect state:

- `_preHarvestAccount(msg.sender)`
- `_poke(msg.sender)`
- `_burnShares(msg.sender, yieldToken, shares)`

[14] The reason why `withdrawUnderlying` uses `_preHarvestAccount` instead of `_preHarvestToken` is to make sure the user does not withdraw enough collateral that a future harvest will leave the CDP undercollateralized. This is not an issue on `liquidate` since it also causes the debt to decrease, so it is possible that some calls to `withdrawUnderlying` will revert when `liquidate` would not.

- `_sync(yieldToken, amountYield, _usub)`
- `_unwrap(yieldToken, amountYield, recipient, params)`

A successful call of `repay(underlyingToken, amountUnderlying, recipient)` makes the following calls that affect state:

- `_poke(recipient)`
- `_updateDebt(recipient, SafeCast.toInt256(_normalizeUnderlying(underlyingToken, amountUnderlying)), _ssub)`
- `TokenUtils.safeTransferFrom(underlyingToken, msg.sender, transmuter, amountUnderlying)`
- `IERC20TokenReceiver(transmuter).onERC20Received(underlyingToken, amountUnderlying)`

Note first that the calls to `burnShares`, `updateDebt` (since `recipient == msg.sender`), `_sync` and `onERC20Received` are the same, and the calls to `_preHarvestAccount` and `_preHarvestToken` have the same effect on the balance of `yieldToken`. Additionally, the calls to `_unwrap` and `safeTransfer/safeTransferFrom` transfer the same amounts. In `liquidate` the Alchemist receives and transfers the unwrapped underlying tokens to the TransmuterBuffer, while in `withdrawUnderlying/repay` the caller does. However, since the amount received and the amount transferred are the same, the end result is the same.

The only remaining differences to address are the different order of operations and the fact that `_poke` is called twice in the second case. The latter can be ignored, since `_poke` is a no-op if the weights have not been updated since the last call of `_poke`, and none of the other operations update the weights. To analyze the effect of the different order of operations, we consider what part of the state each operation affects:

- `_preHarvestToken/_preHarvestAccount` affects the yield token's `balance` and `harvestBuffer`.
- `_unwrap` affects the underlying token balance (either in the Alchemist or the caller).
- `_poke` affects the weights and the CDP's `debt` and `lastAccruedWeight`. It also depends on the CDP's `balance`.
- `_burnShares` affects the CDP's `balance` and the yield token's `totalShares`. It might also remove tokens from the CDP's `depositedTokens`.
- `_updateDebt` affects the CDP's `debt`.
- `_sync` affects the yield token's `balance` and `expectedValue`.
- `TokenUtils.safeTransfer/safeTransferFrom` affects the underlying token balance (either in the Alchemist or the caller, and in the TransmuterBuffer).
- `IERC20TokenReceiver(transmuter).onERC20Received` affects the TransmuterBuffer and Transmuter states.

`_preHarvestToken`/`_preHarvestAccount` and `_sync` both affect the yield token's `balance`. `_unwrap` and `safeTransfer`/`safeTransferFrom` both affect the underlying token balance. `_poke` and `_updateDebt` both affect the CDP's `debt`, and `_poke` also depends on the CDP's `balance` that is updated by `_burnShares`. Since each pair of dependent operations is called in the same order in both scenarios, the effect is the same.

Invariant A7

Invariant A7: Assuming the price of a yield token never drops to 0, the expected value of the yield token equals 0 only if its balance equals 0.

Initially, both the balance and the expected value are 0. There are two ways in which `_yieldTokens[yieldToken].expectedValue`, can be updated.

The first is by the admin function `snap`, which sets the expected value to $b \cdot p$, where b is the balance and p is the current price of the yield token. Since we assume $p > 0$, after this operation the expected value will equal 0 only if the balance equals 0.

The second is via the `_sync` function, which updates the balance from b to $op(b, \Delta)$ and the expected value from e to $op(e, \Delta \cdot p)$, where op is passed as the parameter `op` and is either unsigned addition or subtraction, and Δ is passed as the parameter `delta`. Every function that calls `_sync` first calls `_preHarvestToken`/`_preHarvestAccount`, which updates b to $b - (b \cdot p - e) / p = b - b + e / p = e / p$ if and only if $b \cdot p > e$.

If b is updated to e / p, then $op(b, \Delta) = op(e / p, \Delta) = op(e, \Delta \cdot p) / p$ (since op is either addition or subtraction). Since we assume that $p > 0$, this means that if $op(e, \Delta \cdot p) = 0$ then $op(b, \Delta) = 0$.

If b is not updated, then $b \cdot p \leq e$. If op is addition, since $p > 0$, b and e either both remain unchanged or both increase, depending on whether $\Delta > 0$. Therefore, the invariant is preserved. If op is subtraction and b and e are both 0, then the function reverts unless $\Delta = 0$, in which case both remain unchanged. If op is subtraction and b and e are greater than 0, then $op(e, \Delta \cdot p) = e - \Delta \cdot p$, which means $op(e, \Delta \cdot p) = 0$ iff $e - \Delta \cdot p = 0$ iff $e / p = \Delta$. Therefore in this case $op(b, \Delta) = b - \Delta = b - e / p$. Since $b \cdot p \leq e$, or equivalently $b \leq e / p$, then $op(b, \Delta)$ either reverts or returns 0. Therefore, if the function returns successfully with $op(e, \Delta \cdot p) = 0$ then $op(b, \Delta) = 0$, preserving the invariant.

The only remaining ways to update the balance are via pre-harvest, either by `_preHarvestToken` in `harvest` and `donate`, `_preHarvestAccount` in `mint` and `mintFrom`, or calls of `_preHarvestAccount` when withdrawing other tokens. Since pre-harvest guarantees that $b \leq e / p$, it also guarantees that if $e = 0$ then $b = 0$, preserving the invariant.

Invariant A8

Invariant A8: If a yield token or its underlying token is disabled in the protocol, then no user has any balance in that yield token.

The only function that can increase a user's balance in a yield token is `_issueSharesForAmount`. This function is only called by the `deposit` and `depositUnderlying` external functions, both of which first call `_checkEnabled` on the yield token. Since `_checkEnabled` reverts if either the yield token or its underlying token is disabled, there is no way for the balance of a user in a disabled token to increase from 0. Since the only functions that can disable a token are `disableUnderlyingToken` and `disableYieldToken`, which use the `onlySentinelOrAdmin` modifier, there is no way to have balance on a disabled token if the governance behaves correctly.

Invariant A9

Invariant A9: Assume no loss occurs on yield tokens. Then, every CDP (after being updated by `_poke`) is always "healthy", meaning it maintains at least the minimum collateralization ratio (assuming this ratio is at least 1).

A CDP with no debt or negative debt (credit) is always considered healthy, therefore we assume in the following that the debt is greater than 0.

The collateralization ratio of a CDP is given by v / d, where v is the total value of the CDP's collateral and d is the CDP's debt. The total value v is calculated by $\sum_i (s_i \cdot b_i \cdot p_i / t_i)$, where, for the i-th yield token deposited in the CDP, s_i is the number of shares in the CDP, b_i is the total balance in the Alchemist, p_i is the price in underlying tokens and t_i is the total number of shares in the Alchemist.

We will prove a stronger invariant, namely that $d \cdot r \leq \sum_i (s_i \cdot e_i / t_i)$, where e_i is the expected value of token i and r is the minimum collateralization ratio. Since we assume no loss occurs on a yield token, $e_i \leq b_i \cdot p_i$, and therefore $\sum_i (s_i \cdot e_i / t_i) \leq \sum_i (s_i \cdot b_i \cdot p_i / t_i) = v$. Therefore, $d \cdot r \leq \sum_i (s_i \cdot e_i / t_i) \leq \sum_i (s_i \cdot b_i \cdot p_i / t_i) = v$. Since we assume $d > 0$, this means $r \leq v / d$. Therefore, the new invariant subsumes the original invariant.

The only external functions that can decrease $\sum_i (s_i \cdot e_i / t_i)$ are `snap`, `withdraw`, `withdrawUnderlying`, `withdrawFrom`, `withdrawUnderlyingFrom` and `liquidate` (note that `_issueShares` and the external functions that call it cannot, since they increase s_i and t_i by the same amount; since s_i is always less than or equal to t_i, this can only increase the ratio s_i / t_i).

The four versions of `withdraw` decrease both s_i and t_i by a number of shares a (via the `_burnShares` function) and also decrease e_i by $a \cdot b_i \cdot p_i / t_i$ (via the function `_sync`). However, all four versions call `_preHarvestAccount` before performing this update. For all yield tokens deposited in the CDP, as long as $e_i \leq b_i \cdot p_i$ before being called, `_preHarvestAccount` guarantees that $e_i = b_i \cdot p_i$ on return. Therefore, e_i / t_i gets updated to $(e_i - a \cdot b_i \cdot p_i / t_i) / (t_i - a) = (e_i - a \cdot e_i / t_i) / (t_i - a) = (t_i \cdot e_i - a \cdot e_i) / (t_i \cdot (t_i - a)) = e_i \cdot (t_i - a) / (t_i \cdot (t_i - a)) = e_i / t_i$. In other words, the ratio e_i / t_i does not change.

Therefore, for all CDPs except the one being withdrawn from, the $s_i \cdot e_i / t_i$ remains the same (since the s_i of those CDPs does not get updated). For the CDP being withdrawn from, the $s_i \cdot e_i / t_i$ term gets updated to $(s_i - a) \cdot e_i / t_i$. This can technically make the sum smaller than $d \cdot r$. However, all four versions of `withdraw` call `_validate` at the end, which reverts if it is not true that $d \cdot r \leq \sum_i (s_i \cdot b_i \cdot p_i / t_i)$. Recall that `_preHarvestAccount` makes $e_i = b_i \cdot p_i$ for all deposited yield tokens. For the withdrawn token, while e_i decreases to $e_i - a \cdot e_i / t_i$, `_sync` also updates b_i to $b_i - a \cdot b_i / t_i = (b_i \cdot p_i - a \cdot b_i \cdot p_i / t_i) / p_i = (e_i - a \cdot e_i / t_i) / p_i$. Therefore, when `_validate` is called, it is still true that $e_i = b_i \cdot p_i$, and therefore $s_i \cdot b_i \cdot p_i / t_i = s_i \cdot e_i / t_i$. Since this holds for all deposited yield tokens, `_validate` will revert unless $d \cdot r \leq \sum_i (s_i \cdot e_i / t_i)$ (using the updated values).

The `liquidate` function calls `_preHarvestToken`, which makes $e_i = b_i \cdot p_i$ for the yield token being liquidated, and then performs the same updates as the `withdraw` function without calling `_validate`. However, it also decreases the debt of the CDP by $a \cdot b_i \cdot p_i / t_i = a \cdot e_i / t_i$ (by calling `_updateDebt`). This means that $d \cdot r$ decreases by $r \cdot a \cdot e_i / t_i$. Since a single $s_i \cdot e_i / t_i$ term decreases to $(s_i - a) \cdot e_i / t_i = (s_i \cdot e_i / t_i) - (a \cdot e_i / t_i)$, the entire sum $\sum_i (s_i \cdot e_i / t_i)$ decreases by $a \cdot e_i / t_i$, which is less than or equal to $r \cdot a \cdot e_i / t_i$ since we assume that $r \geq 1$. Since initially $d \cdot r \leq \sum_i (s_i \cdot e_i / t_i)$, and the left-hand side decreased at least as much as the right-hand side, the inequality is maintained. Therefore, the invariant is preserved.

Finally, since we assume no losses, meaning $e_i \leq b_i \cdot p_i$, and `snap` sets $e_i = b_i \cdot p_i$, this can only increase the sum $\sum_i (s_i \cdot e_i / t_i)$.

We now consider the external functions that may increase d: `mint` and `mintFrom`. Both these functions call `_preHarvestAccount` at the beginning and don't update e_i and b_i in any other way. Therefore, at the end $e_i = b_i \cdot p_i$ for all yield tokens, and as a consequence $\sum_i (s_i \cdot e_i / t_i) = \sum_i (s_i \cdot b_i \cdot p_i / t_i) = v$. Since both call `_validate` after making the updates, the function reverts unless $d \cdot r \leq v = \sum_i (s_i \cdot e_i / t_i)$.

Since r is a parameter controlled by governance, which we assume will not behave in a way that violates the invariant, we have accounted for all other ways that the inequality $d \cdot r \leq \sum_i (s_i \cdot e_i / t_i)$ can change. Therefore, under our assumptions the invariant is preserved.

C06: Minting tokens is properly privileged

The only way to mint synthetic tokens is through the `mint` function, and, temporarily, through flash loans. The only addresses that can call `mint` are those which have been whitelisted by the admin. The admin is expected to only assign this privilege only to Alchemists. Based on the general trust in the integrity of the admin account, multisig or DAO contract, we can conclude that there is no way to mint tokens illegitimately, other than by an error in an Alchemist which would allow token minting.

An Alchemist can only call `mint` on the AlchemicToken contract via the Alchemist's own `mint` function. This function is external, only mints the exact amount requested, and cannot bypass the debt limit of the CDP or the minting limit of the given Alchemist.

C07: Flash loans can’t drive the Alchemist to arithmetic limits

The `maximumExpectedValue` protects against situations where an attacker would borrow an astronomical number of underlying tokens through a flash mint and trigger arithmetic edge cases in the `uint256` values. Thus we can assume that the amount of tokens deposited of any kind is always "reasonable", i.e. within orders of magnitude of actual circulating supply.

Appendix 1: Token types per entity

	Alchemist	**TransmuterBuffer**	**Transmuters**	**User**
Holds token	Yield tokens	Underlying tokens	Synthetic tokens	Yield tokens Underlying tokens Synthetic tokens

Possible Actions	**Alchemist**	**TransmuterBuffer**	**Transmuters**	**User**
Mint synthetic	x			
Burn synthetic	x		x	x