Alchemix contest
Findings & Analysis Report

2022-07-18

Table of contents

- L-09 Missing checks for `address(0x0)` when assigning values to `address` state variables
- L-10 `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`
- L-11 Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions
- N-01 Adding a `return` statement when the function defines a named return variable, is redundant
- N-02 `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings
- N-03 `public` functions not called by the contract should be declared `external` instead
- N-04 `2**<n> - 1` should be re-written as `type(uint<n>).max`
- N-05 `constant` s should be defined rather than using magic numbers
- N-06 Redundant cast
- N-07 Numeric values having to do with time should use time units for readability
- N-08 Missing event for critical parameter change
- N-09 Use a more recent version of solidity
- N-10 Use a more recent version of solidity
- N-11 Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)
- N-12 Inconsistent spacing in comments
- N-13 Non-library/interface files should use fixed compiler versions, not floating ones
- N-14 Typos
- N-15 File does not contain an SPDX Identifier
- N-16 File is missing NatSpec
- N-17 NatSpec is incomplete
- N-18 Event is missing `indexed` fields

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Alchemix smart contract system written in Solidity. The audit contest took place between May 5—May 18 2022.

Wardens

74 Wardens contributed reports to the Alchemix contest:

1. hyh
2. AuditsAreUS
3. TerrierLover
4. WatchPug (jtp and ming)
5. cccz
6. Ruhum
7. 0x52
8. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
9. 0x1337
10. dirk_y
11. 0xsomeone
12. IIIIIII
13. tintin
14. GimelSec (rayn and sces60107)
15. BowTiedWardens (BowTiedHeron and BowTiedPickle and m4rio_eth and Dravee and BowTiedFirefox)
16. joestakey
17. 0x1f8b

18. 0xDjango
19. mics
20. 0xkatana
21. fatherOfBlocks
22. robee
23. 0x4non
24. simon135
25. 0xNazgul
26. samruna
27. MaratCerby
28. csanuragjain
29. sikorico
30. horsefacts
31. catchup
32. ellahi
33. oyc_109
34. MiloTruck
35. Funen
36. throttle
37. Cityscape
38. bobirichman
39. Hawkeye (0xwags and 0xmint)
40. Waze
41. hake
42. kenta
43. JC
44. shenwilly
45. AlleyCat
46. jayjonah8

47. Picodes
48. cryptphi
49. BouSalman
50. delfin454000
51. kebabsec (okkothejawa and FlameHorizon)
52. sashik_eth
53. 0xf15ers (remora and twojoy)
54. Tomio
55. _Adam
56. hansfriese
57. ignacio
58. 0v3rf10w
59. Fitraldys
60. Randyyy
61. UnusualTurtle
62. augustg

This contest was judged by 0xleastwood.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 17 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 46 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 46 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Alchemix contest repository, and is composed of 27 smart contracts written in the Solidity programming language and includes 7,000 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (17)

[M-01] Alchemist can mint `AlTokens` above their assigned ceiling by calling `lowerHasMinted()`

Submitted by tintin, also found by 0xsomeone, AuditsAreUS, and hyh

AlchemicTokenV2Base.sol#L111-L124
AlchemicTokenV2Base.sol#L189-L191

An alchemist / user can mint more than their alloted amount of AlTokens by calling `lowerHasMinted()` before they reach their minting cap.

Proof of Concept

Function `mint()` in `AlchemicTokenV2Base.sol`

```
function mint(address recipient, uint256 amount) external only
  if (paused[msg.sender]) {
    revert IllegalState();
  }

  uint256 total = amount + totalMinted[msg.sender];
  if (total > mintCeiling[msg.sender]) {
    revert IllegalState();
  }

  totalMinted[msg.sender] = total;

  _mint(recipient, amount);
}
```

Note the require conditional check that `total > mintCeiling[msg.sender]`.

In the same contract, there is the function `lowerHasMinted()` with the same permission level as mint and is thus callable by the same user as well.

```
function lowerHasMinted(uint256 amount) external onlyWhitelist
  totalMinted[msg.sender] = totalMinted[msg.sender] - amount;
}
```

It is clear that a user can accumulate an infinite (within supply) amount of AlTokens by calling `lowerHasMinted()` before any action that would make them exceed their minting cap.

Tools Used

Manual review, VScode

Recommended Mitigation Steps

Change the permissioning on `lowerHasMinted()` to be restricted to a higher permissioned role like `onlySentinel()` , or deprecate this function as I could not find any uses of it throughout the codebase or in tests.

0xfoobar (Alchemix) confirmed

0xleastwood (judge) commented:

> Great find! This would allow whitelisted account to mint any number of tokens. However, as this pertains to only whitelisted accounts, I think `medium` severity is justified and correct.

[M-02] TransmuterBuffer.sol calls depositUnderlying with no slippage bounds

Submitted by 0x52

Loss of funds in TransmuterBuffer

Proof of Concept

If the buffer is called during and unfavorable time then a large portion of deposited funds may be lost due to slippage because deposit is called with 0 as the minimum out allowing any level of slippage

Recommended Mitigation Steps

Implement a slippage calculation similar to `\_alchemistWithdraw` to protect against it

0xfoobar (Alchemix) acknowledged, disagreed with severity and commented:

> This function is only called by keeper bots harvesting yields, which should not be subject to large slippage and could be sent through a private mempool if necessary. However, we acknowledge that a configurable parameter could enable greater protection, even if in practice the issue does not occur.

Oxleastwood (judge) decreased severity to Medium and commented:

> Because this requires the keeper role to sandwich attack the protocol when yield is harvested, this better fits the criteria of a `medium` severity issue.

[M-03] DoS in wrap and unwrap

Submitted by CertoraInc

FuseTokenAdapterV1.sol#L76
FuseTokenAdapterV1.sol#L98

The code is doing wrong check, so when things will work it will revert.

Proof of Concept

In the function `wrap()` there is this lines:

```
        if ((error = ICERC20(token).mint(amount)) != NO_ERROR) {
            revert FuseError(error);
        }
```

but `mint` returns the amount that minted, so when `error = amount` the check will fail even though it worked good.

Same in `unwrap`:

```
    if ((error = ICERC20(token).redeem(amount)) != NO_ERROR) {
            revert FuseError(error);
        }
```

the redeem returns the amount.

Recommended Mitigation Steps

I recommend to change the lines like this: in wrap: `if ((error = ICERC20(token).mint(amount)) != amount) { revert FuseError(error); }` and in unwrap: `if ((error = ICERC20(token).redeem(amount)) != amount) { revert FuseError(error); }`

0xfoobar (Alchemix) confirmed, disagreed with severity and commented:

> This would not cause any loss of user funds because the deposit function would revert, but it is a needed fix in the Fuse Adapter. So recommend a lower severity.

0xleastwood (judge) decreased severity to Medium and commented:

> As no assets are at risk, medium risk seems correct because only the availability of the protocol is impacted.

[M-04] YearnTokenAdapter allows a maximum loss of 100% when withdrawing

Submitted by Ruhum

YearnTokenAdapter.sol#L13
YearnTokenAdapter.sol#L43

YearnTokenAdapter allows slippage of 100% when withdrawing from the vault which will cause a loss of funds.

Here's the documentation straight from the vault contract:
https://github.com/yearn/yearn-vaults/blob/main/contracts/Vault.vy#L1025-L1073
It allows the user to specify the `maxLoss` as the last parameter. It determines how many shares can be burned to fulfill the withdrawal. Currently, the contract uses 10.000 which is 100%. Meaning there is no slippage check at all. This is bound to cause a loss of funds.

I'd suggest letting the user determine the slippage check themselves instead of hardcoding it.

Proof of Concept

Current `maxLoss` value: https://github.com/code-423n4/2022-05-alchemix/blob/main/contracts-full/adapters/yearn/YearnTokenAdapter.sol#L13

call to Yearn vault's `withdraw()` function: https://github.com/code-423n4/2022-05-alchemix/blob/main/contracts-full/adapters/yearn/YearnTokenAdapter.sol#L43

Recommended Mitigation Steps

Allow the user to specify the slippage value themselves:

```
function unwrap(uint256 amount, address recipient, uint maxI
    TokenUtils.safeTransferFrom(token, msg.sender, address(t

    uint256 balanceBefore = TokenUtils.safeBalanceOf(token,

    uint256 amountWithdrawn = IYearnVaultV2(token).withdraw(

    uint256 balanceAfter = TokenUtils.safeBalanceOf(token, a

    // If the Yearn vault did not burn all of the shares the
    // performed by the system because the system always exp
    // this sometimes does not happen in cases where strateg
    // example strategy where this can occur is with Compour
    // they were lent out).
    if (balanceBefore - balanceAfter != amount) {
        revert IllegalState();
    }

    return amountWithdrawn;
}
```

If you don't want to change the ITokenAdapter interface you can also leave the value blank. The vault will then use the default value (`0.01%`)

Oxfoobar (Alchemix) acknowledged, disagreed with severity and commented:

> This could be made more configurable by the end user but yearn vaults do not frequently experience high slippage. Slippage is handled upstream in the

> Alchemist contract. The reason why this slippage is set to 100% is so to permit handling of slippage in the Alchemist for all cases.

0xleastwood (judge) decreased severity to Medium and commented:

> Because we can't know how the yearn strategy implements withdrawals, its possible that it might contain custom swap logic which exposes itself to sandwich attacks. However, at face value, the current use of `MAXIMUM_SLIPPAGE` allows the contract to successfully unwrap their tokens under poor network conditions, but it makes sense for the user to have more control over this. Downgrading this to medium risk as I believe it is more in line with that.

[M-05] No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision

Submitted by 0x1337

AlchemicTokenV2Base.sol#L20
CrossChainCanonicalBase.sol#L12
TransmuterV2.sol#L26
CrossChainCanonicalAlchemicTokenV2.sol#L7

For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments" (quote OpenZeppelin). Otherwise it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences to the child contracts, potentially causing loss of user fund or cause the contract to malfunction completely.

Refer to the bottom part of this article: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable

Proof of Concept

Several contracts are intended to be upgradeable contracts in the code base, including

- AlchemicTokenV2Base
- CrossChainCanonicalBase
- CrossChainCanonicalAlchemicTokenV2
- TransmuterV2

However, none of these contracts contain storage gap. The storage gap is essential for upgradeable contract because “It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments”. Refer to the bottom part of this article:

https://docs.openzeppelin.com/contracts/3.x/upgradeable

As an example, both the `AlchemicTokenV2Base` and the `CrossChainCanonicalBase` are intended to act as the base contracts in the project. If the contract inheriting the base contract contains additional variable, then the base contract cannot be upgraded to include any additional variable, because it would overwrite the variable declared in its child contract. This greatly limits contract upgradeability.

Recommended Mitigation Steps

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

0xfoobar (Alchemix) confirmed

0xleastwood (judge) commented:

> Agree with warden and severity. Storage gaps are essential wherever inheritance is used by an upgradeable contract.

[M-06] EthAssetManager and ThreePoolAssetManager don’t control Meta tokens decimals

Submitted by hyh

Both contracts treat meta assets as if they have fixed decimals of 18. Minting logic breaks when it's not the case. However, meta tokens decimals aren't controlled.

If actual meta assets have any other decimals, minting slippage control logic of both contracts will break up as `total` is calculated as a plain sum of token amounts.

In the higher token decimals case `minTotalAmount` will be magnitudes higher than actual amount Curve can provide and minting becomes unavailable.

In the lower token decimals case `minTotalAmount` will lack value and slippage control will be rendered void, which opens up a possibility of a fund loss from the excess slippage.

Setting severity to medium as the contract can be used with various meta tokens (`_metaPoolAssetCache` can be filled with any assets) and, whenever decimals differ from 18 `add_liquidity` uses, its logic be broken: the inability to mint violates the contract purpose, the lack of slippage control can lead to fund losses.

I.e. this is system breaking impact conditional on a low probability assumption of different meta token decimals.

Proof of Concept

Meta tokens decimals are de facto hard coded into the contract as plain amounts are used (L. 905):

[ThreePoolAssetManager.sol#L896-L905]

```
function _mintMetaPoolTokens(
    uint256[NUM_META_COINS] calldata amounts
) internal returns (uint256 minted) {
    IERC20[NUM_META_COINS] memory tokens = _metaPoolAssetCac

    uint256 total = 0;
    for (uint256 i = 0; i < NUM_META_COINS; i++) {
        if (amounts[i] == 0) continue;
```

```
            total += amounts[i];
```

[ThreePoolAssetManager.sol#L915-L919]

```
        uint256 expectedOutput    = total * CURVE_PRECISION / me
        uint256 minimumMintAmount = expectedOutput * metaPoolSli

        // Add the liquidity to the pool.
        minted = metaPool.add_liquidity(amounts, minimumMintAmou
```

The same plain sum approach is used in EthAssetManager._mintMetaPoolTokens:

[EthAssetManager.sol#L566-L573]

```
        uint256 total = 0;
        for (uint256 i = 0; i < NUM_META_COINS; i++) {
            // Skip over approving WETH since we are directly sw
            if (i == uint256(MetaPoolAsset.ETH)) continue;

            if (amounts[i] == 0) continue;

            total += amounts[i];
```

When this decimals assumption doesn't hold, the slippage logic will not hold too: either the mint be blocked or slippage control disabled.

Notice, that ThreePoolAssetManager.calculateRebalance do query alUSD decimals (which is inconsistent with the above as it's either fix and control on inception or do not fix and accommodate the logic):

[ThreePoolAssetManager.sol#L338-L338]

```
    decimals     = SafeERC20.expectDecimals(address(alUSD));
```

Recommended Mitigation Steps

If meta assets are always supposed to have fixed decimals of 18, consider controlling it at the construction time.

I.e. the decimals can be controlled in constructors:

EthAssetManager.sol#L214-L219

```
            for (uint256 i = 0; i < NUM_META_COINS; i++) {
                _metaPoolAssetCache[i] = params.metaPool.coins(i);
                if (_metaPoolAssetCache[i] == IERC20(0xEeeeeEeeeEeEe
                    _metaPoolAssetCache[i] = weth;
+               } else {
+                   // check the decimals
                                        }
            }
```

ThreePoolAssetManager.sol#L254-L256

```
            for (uint256 i = 0; i < NUM_META_COINS; i++) {
                _metaPoolAssetCache[i] = params.metaPool.coins(i);
+               // check the decimals
            }
```

In this case further decimals reading as it's done in calculateRebalance() is redundant.

Otherwise (which is less recommended as fixed decimals assumption is viable and simplify the logic) the meta token decimals can be added to calculations similarly to stables:

ThreePoolAssetManager.sol#L779-L779

```
    normalizedTotal += amounts[i] * 10**missingDecimals;
```

0xfoobar (Alchemix) confirmed

0xleastwood (judge) commented:

> Agree with issue and its severity. `minTotalAmount` is affected by a change in a token's decimals, leading to improper handling by the contract.

[M-07] AutoleverageBase: Must approve 0 first

Submitted by cccz

Some tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value.They must first be approved by zero and then the actual allowance must be approved.

Proof of Concept

[AutoleverageBase.sol#L61-L63]

[AutoleverageBase.sol#L147-L147]

[AutoleverageBase.sol#L178-L179]

Recommended Mitigation Steps

```
function approve(address token, address spender) internal {
+  IERC20(token).approve(spender, 0);
    IERC20(token).approve(spender, type(uint256).max);
}
```

Oxtao (Alchemix) confirmed and commented:

> This implementation will not work for tokens like USDT where the approval is not set to `0` initially

Oxleastwood (judge) commented:

> It seems like `approve()` will fail to execute on non-standard tokens which require the approval amount to start from zero. This is valid and should be updated to handle such tokens.

[M-08] YearnTokenAdapter's wrap can become stuck as it uses one step approval for an arbitrary underlying

Submitted by hyh

Some tokens do not allow for approval of positive amount when allowance is positive already (to handle approval race condition, most known example is USDT).

This can cause the function to stuck whenever a combination of such a token and leftover approval be met. The latter can be possible if, for example, yearn vault is becoming full on a particular wrap() call and accepts only a part of amount, not utilizing the approval fully.

Then each next safeApprove will revert and wrap becomes permanently unavailable. Setting the severity to medium as depositing (wrapping) is core functionality for the contract and its availability is affected.

Proof of Concept

wrap use one step approve:

YearnTokenAdapter.sol#L30-L32

```
        function wrap(uint256 amount, address recipient) external ov
            TokenUtils.safeTransferFrom(underlyingToken, msg.sender,
            TokenUtils.safeApprove(underlyingToken, token, amount);
```

Some ERC20 forbid the approval of positive amount when the allowance is positive:

https://github.com/d-xo/weird-erc20#approval-race-protections

For example, USDT is supported by Yearn and can be the underlying asset:

https://yearn.finance/#/vault/0x7Da96a3891Add058AdA2E826306D812C638D87a7

Recommended Mitigation Steps

As the most general approach consider approving zero before doing so for the amount:

```
    function wrap(uint256 amount, address recipient) external ov
        TokenUtils.safeTransferFrom(underlyingToken, msg.sender,
+       TokenUtils.safeApprove(underlyingToken, token, 0);
        TokenUtils.safeApprove(underlyingToken, token, amount);
```

Oxfoobar (Alchemix) confirmed

Oxleastwood (judge) commented:

> It seems like `approve()` will fail to execute on non-standard tokens which require the approval amount to start from zero. This is valid and should be updated to handle such tokens.

[M-09] TransmuterBuffer's setAlchemist will freeze deposited funds

Submitted by hyh

Currently setAlchemist doesn't check whether there are any open positions left with the old Alchemist before switching to the new one.

As this require a number of checks the probability of operational mistake isn't low and it's prudent to introduce the main controls directly to the code to minimize it. In the case if the system go on with new Alchemist before realizing that there are some funds left in the old one, tedious and error prone manual recovery will be needed. There is also going to be a corresponding reputational damage.

Setting the severity to medium as while the function is admin only, the impact is up to massive user fund freeze, i.e. this is system breaking with external assumptions.

Proof of Concept

Alchemist implementation change can happen while there are open deposits remaining with the current contract. As there looks to be no process to transfer them

in the code, such TransmuterBuffer's funds will be frozen with old alchemist:

[TransmuterBuffer.sol#L230-L232]

```
        function setAlchemist(address _alchemist) external override
            sources[alchemist] = false;
            sources[_alchemist] = true;
```

Recommended Mitigation Steps

Consider requiring that all exposure to the old Alchemist is closed, for example both `getAvailableFlow` and `getTotalCredit` is zero.

[TransmuterBuffer.sol#L230-L231]

```
        function setAlchemist(address _alchemist) external override
    +                   require(getTotalCredit() == 0, "Credit exists
    +           for (uint256 j = 0; j < registeredUnderlyings.length; j+
    +               require(getTotalUnderlyingBuffered[registeredUnderly
    +           }
            sources[alchemist] = false;
```

[Oxfoobar (Alchemix) confirmed]

[Oxleastwood (judge) commented]:

> This is useful in preventing loss of funds when changing the protocol's alchemist contract.

[M-10] New gALCX token denomination can be depressed by the first depositor

Submitted by hyh, also found by 0xsomeone

An attacker can become the first depositor for a recently created gALCX contract, providing a tiny amount of ALCX tokens by calling `stake(1)` (raw values here, `1` is

`1 wei`, `1e18` is `1 ALCX`). Then the attacker can directly transfer, for example, `10^6 * 1e18 - 1` of ALCX to the gALCX contract and run bumpExchangeRate(), effectively setting the cost of `1` gALCX to be `10^6 * 1e18` of ALCX. The attacker will still own 100% of the gALCX's ALCX pool being the only depositor.

All subsequent depositors will have their ALCX token investments rounded to `10^6 * 1e18`, due to the lack of precision which initial tiny deposit caused, with the remainder divided between all current depositors, i.e. the subsequent depositors lose value to the attacker.

For example, if the second depositor brings in `1.9*10^6 * 1e18` of ALCX, only `1` of new vault to be issued as `1.9*10^6 * 1e18` divided by `10^6 * 1e18` will yield just `1`, which means that `2.9*10^6 * 1e18` total ALCX pool will be divided 50/50 between the second depositor and the attacker, as each will have `1 wei` of the total `2 wei` of vault tokens, i.e. the depositor lost and the attacker gained `0.45 * 10^6 * 1e18` of ALCX tokens.

As there are no penalties to exit with gALCX.unstake(), the attacker can remain staked for an arbitrary time, gathering the share of all new deposits' remainder amounts.

Placing severity to be medium as this is principal funds loss scenario for many users (most of depositors), easily executable, but only new gALCX contract instances are vulnerable.

Proof of Concept

gAmount of gALCX to be minted is determined as a quotient of `amount` provided and `exchangeRate`:

gALCX.sol#L93-L94

```
uint gAmount = amount * exchangeRatePrecision / exchange
_mint(msg.sender, gAmount);
```

gALCX.sol#L15

```
uint public constant exchangeRatePrecision = 1e18;
```

`exchangeRate` accumulates balance increments relative to total gALCX supply:

[gALCX.sol#L69-L76]

```
function bumpExchangeRate() public {
    // Claim from pool
    pools.claim(poolId);
    // Bump exchange rate
    uint balance = alcx.balanceOf(address(this));

    if (balance > 0) {
        exchangeRate += (balance * exchangeRatePrecision) /
```

When gALCX contract is new, the very first stake -> bumpExchangeRate() yields nothing as the balance is empty, i.e. `exchangeRate` is still `1e18` and `gAmount == amount`:

[gALCX.sol#L85-L93]

```
function stake(uint amount) external {
    // Get current exchange rate between ALCX and gALCX
    bumpExchangeRate();
    // Then receive new deposits
    bool success = alcx.transferFrom(msg.sender, address(thi
    require(success, "Transfer failed");
    pools.deposit(poolId, amount);
    // gAmount always <= amount
    uint gAmount = amount * exchangeRatePrecision / exchange
```

This way, as there is no minimum amount or special treatment for the first deposit, the very first `gAmount` can be made `1 wei` with `stake(1)` call.

Then, a combination of direct ALCX transfer and bumpExchangeRate() will make `exchangeRate` equal to the total amount provided by the attacker, say `10^6 * 1e18 * 1e18`, as totalSupply is `1 wei`.

When a second depositor enters, the amount of gALCX to be minted is calculated as `amount * exchangeRatePrecision / exchangeRate`, which is `amount / (10^6 * 1e18)`, which will trunk the `amount` to the nearest divisor of `10^6 * 1e18`, effectively dividing the remainder between the depositor and the attacker.

For example, if the second depositor brings in `1.9*10^6` ALCX, only `1` (`1 wei`) of gALCX to be issued as `1.9*10^6 * 1e18 * 1e18 / (10^6 * 1e18 * 1e18) = 1`.

As the attacker and depositor both have `1` of gALCX, each owns `(2.9 / 2)*10^6 * 1e18 = 1.45*10^6 * 1e18`, so the attacker effectively stole `0.45*10^6 * 1e18` from the depositor.

Any deposit lower than total attacker's stake, `10^6 * 1e18`, will be fully stolen from the depositor as `0` gALCX tokens will be issued in this case.

References

The issue is similar to the `TOB-YEARN-003` one of the Trail of Bits audit of Yearn Finance:

https://github.com/yearn/yearn-security/tree/master/audits/20210719_ToB_yearn_vaultsv2

Recommended Mitigation Steps

A minimum for deposit value can drastically reduce the economic viability of the attack. I.e. stake() can require each amount to surpass the threshold, and then an attacker would have to provide too big direct investment to capture any meaningful share of the subsequent deposits.

An alternative is to require only the first depositor to freeze big enough initial amount of liquidity. This approach has been used long enough by various projects, for example in Uniswap V2:

Uniswap/UniswapV2Pair.sol#L119-L121

0xfoobar (Alchemix) acknowledged, disagreed with severity and commented:

> Not a risk with current >400 tokenholders, but good to incorporated into future designs.

0xleastwood (judge) commented:

> I think from the perspective of the contest, it is fair to assume that contracts are somewhat fresh. I'd be inclined to keep this as medium because it outlines a viable attack path that should be made public.

[M-11] [gALCX.sol] Attacker can make the contract unusable when totalSupply is 0

Submitted by TerrierLover

An attacker can make the contract unusable when totalSupply is 0. Specifically, `bumpExchangeRate` function does not work correctly which results in making `stake`, `unstake` and `migrateSource` functions that do not work as expected.

Proof of Concept

Here are steps on how the `gALCX` contract can be unusable.

1. `gALCX` contract is deployed
2. The attacker sends the `ALCX` token to the deployed `gALCX` contract directly instead of using `stake` function so that the following `balance` variable has value.

gALCX.sol#L73-L75

uint balance = alcx.balanceOf(address(this));

```
if (balance > 0) {
```

3. Since the `ALCX` token is given to the `gALCX` contract directly, `totalSupply == 0` and `alcx.balanceOf(address(this)) > 0` becomes true.

gALCX.sol#L76

```
exchangeRate += (balance * exchangeRatePrecision) / totalSupply;
```

4. Non attackers try to call `stake` function, but `bumpExchangeRate` function fails because of `(balance * exchangeRatePrecision) / totalSupply` when totalSupply is 0.
5. Owner cannot call `migrateSource` function since `bumpExchangeRate` will be in the same situation mentioned in the step4 above

Recommended Mitigation Steps

Add handling when `totalSupply` is 0 but `alcx.balanceOf(address(this))` is more than 0.

0xfoobar (Alchemix) acknowledged and commented:

> Given that the gALCX deployment has 412 unique tokenholders on mainnet, this series of events is extraordinarily unlikely to occur. But we will keep it in mind for future deployments.

0xleastwood (judge) commented:

> Nice find! Early stakers can DoS new contract deployments, making it impossible for other users to participate in the protocol. As this does not lead to lost funds and is recoverable through redeployment, I believe medium severity to be justified by the warden.

[M-12] registerAsset misuse can permanently disable TransmuterBuffer and break the system

Submitted by hyh

TransmuterBuffer's refreshStrategies() is the only way to actualize _yieldTokens array. The function requires registeredUnderlyings array to match current Alchemist's _supportedUnderlyingTokens. In the same time registeredUnderlyings can be only increased via registerAsset(): there is no way to reduce, remove or reconstruct the array.

This way if registerAsset() was mistakenly called extra time or alchemist was switched with setAlchemist to a new one with less supported assets, then the strategy refresh becomes impossible and the TransmuterBuffer be blocked as it cannot be properly used without synchronization with Alchemist.

The redeployment of the contract doesn't provide an easy fix as it holds the accounting data that needs to be recreated (flowAvailable, currentExchanged mappings).

Proof of Concept

refreshStrategies require registeredUnderlyings to be equal to Alchemist's supportedUnderlyingTokens:

[TransmuterBuffer.sol#L377-L379]

```
if (registeredUnderlyings.length != supportedUnderlyingT
    revert IllegalState();
}
```

If registeredUnderlyings has length more than Alchemist's _supportedUnderlyingTokens it doesn't look to be fixable and prohibits the future use of the contract, i.e. breaks the system.

Recommended Mitigation Steps

The issue is that there is no way to unregister the asset, so consider introducing a function to remove the underlying or simply delete the array so it can be reconstructed with a sequence of registerAsset calls.

thetechnocratic (Alchemix) acknowledged and commented:

> There is no way for `registerAsset` to be accidentally called too many times, and it reverts if an asset doesn't exist in the Alchemist or has already been registered. The TransmuterBuffer *could* be assigned a new Alchemist with fewer assets, but it is safe to assume that the operator will not make such a grand oversight.

0xleastwood (judge) commented:

> Useful mitigation to prevent the TransmuterBuffer from being assigned a new Alchemist with fewer assets. In this event, the availability of the protocol is impacted. Valid medium.

[M-13] TransmuterBuffer's _alchemistWithdraw use hard coded slippage that can lead to user losses

Submitted by hyh

exchange() -> _exchange() -> _alchemistWithdraw() is user funds utilizing call sequence and the slippage hard coded to 1% there can cause a range of issues.

For example, if there is not enough shares, the number of shares to withdraw will be unconditionally reduced to the number of the shares available. This can pass under 1% slippage and user will give away up to 1% without giving a consent to such a fee, which is big enough to notice.

On the other hand, in a similar situation when there is not enough shares available a user might knowingly want to execute with even bigger fee, but hard coded slippage will not be met and the withdraw be unavailable and funds frozen.

Setting the severity to medium as the end impact is either modest user fund loss or exchange functionality unavailability.

Proof of Concept

_alchemistWithdraw uses hard coded 1% slippage threshold and rewrites wantShares to be availableShares once TransmuterBuffer's position isn't big enough:

[TransmuterBuffer.sol#L511-L524]

```
function _alchemistWithdraw(address token, uint256 amountUnc
    uint8 decimals = TokenUtils.expectDecimals(token);
    uint256 pricePerShare = IAlchemistV2(alchemist).getUnder
    uint256 wantShares = amountUnderlying * 10**decimals / p
    (uint256 availableShares, uint256 lastAccruedWeight) = I
    if (wantShares > availableShares) {
        wantShares = availableShares;
```

```
        }
        // Allow 1% slippage
        uint256 minimumAmountOut = amountUnderlying - amountUnde
        if (wantShares > 0) {
            IAlchemistV2(alchemist).withdrawUnderlying(token, wa
        }
    }
```

Alchemist's _unwrap will revert withdrawUnderlying call once minimumAmountOut isn't met:

[AlchemistV2.sol#L1344-L1346]

```
        if (amountUnwrapped < minimumAmountOut) {
            revert SlippageExceeded(amountUnwrapped, minimumAmou
        }
```

There are 2 use cases for the function:

exchange (onlyKeeper) -> _exchange -> _alchemistWithdraw,

setFlowRate (onlyAdmin) -> _exchange -> _alchemistWithdraw

exchange() is the most crucial as it should be able to fulfil various types of user funds exchange requests:

[TransmuterBuffer.sol#L526-L546]

```
    /// @notice Pull necessary funds from the Alchemist and exch
    ///
    /// @param underlyingToken The underlying-token to exchange.
    function _exchange(address underlyingToken) internal {
        _updateFlow(underlyingToken);

        uint256 totalUnderlyingBuffered = getTotalUnderlyingBuff
        uint256 initialLocalBalance = TokenUtils.safeBalanceOf(u
        uint256 want = 0;
        // Here we assume the invariant underlyingToken.balanceO
        if (totalUnderlyingBuffered < flowAvailable[underlyingTo
```

```
            // Pull the rest of the funds from the Alchemist.
            want = totalUnderlyingBuffered - initialLocalBalance
        } else if (initialLocalBalance < flowAvailable[underlyir
            // totalUnderlyingBuffered > flowAvailable so we hav
            want = flowAvailable[underlyingToken] - initialLocal
        }

        if (want > 0) {
            _alchemistAction(want, underlyingToken, _alchemistWi
        }
```

This way, one issue here is that user can end up giving away the full 1% unconditionally to market situation because there are not enough shares available.

Another one is that knowing that the conditions are bad or that there are not enough shares available and willing to run the exchange with bigger slippage the user will not be able to as there are no means to control it and the functionality will end up unavailable, being reverted by Alchemist's _unwrap check.

Recommended Mitigation Steps

Consider adding the function argument with a default value of 1%, so the slippage can be tuned when it is needed.

thetechnocratic (Alchemix) acknowledged and commented:

> Allowing for a caller-defined slippage would enable more flexibility when using the `exchange()` and `setFlowRate()` calls. However, the possibility of needing this flexibility at this time is very small, and because these functions are run by admins/keepers, there is room to modify the code if and when the flexibility becomes required.

Oxleastwood (judge) commented:

> Agree with warden. During periods of high volatility, assets will be locked within the contract. As this limits protocol availability, potentially leading to further loss of funds as users cannot freely exit the protocol and sell tokens, medium risk is justified.

[M-14] A well financed attacker could prevent any other users from minting synthetic tokens

*Submitted by dirky_*

AlchemistV2.sol#L676-L683
AlchemistV2.sol#L704
Limiters.sol#L84

In the `AlchemistV2` contract, users can deposit collateral to then borrow/mint the synthetic tokens offered by the protocol. The protocol also defines a minting limit that specifies how many synthetic tokens can be minted in a given time period. This exists to prevent unbounded minting of synthetic tokens.

Every time a user calls `mint`, the internal `_mint` method decreases the current mint limit. This works as intended. However, there is nothing stopping an attacker from immediately burning their synthetic tokens by calling `burn` and then calling `mint` again. This is possible because the debt position is updated during the `burn` phase, which lets the user then mint again against the same deposited collateral.

In most cases this probably wouldn't be a problem if the mint limit is sufficiently high. However, it is currently possible for a well financed attacker to grief the contract by repeatedly minting and burning synthetic tokens to keep the contract pegged at the mint limit. This will prevent any normal users from minting any synthetic tokens, and hence prevents the protocol from performing as it should.

Proof of Concept

An attacker can repeatedly call `mint` followed by `burn` after depositing some collateral with `deposit`. If this is appropriately sized and timed, it can cause the `mint` call to fail for another user due to the check here that is called during `mint` here.

Tools Used

VSCode

Recommended Mitigation Steps

There should be an additional method added to the `Limiters` library that can increment the mint limit. This method can then be called during a `burn` call in the `AlchemistV2` contract.

```
function increase(LinearGrowthLimiter storage self, uint256 amou
  uint256 value = self.get();
  self.lastValue = value + amount > self.maximum ? self.maximum
}
```

0xfoobar (Alchemix) disputed and commented:

> If somebody griefed, let alone the insanely high capital requirements, governance could simply raise the mint limit.

0xleastwood (judge) commented:

> As far as I can tell, this seems like a valid griefing attack. Assuming no fees are charged on mint/burn actions, it would be viable to use a flash loan to use up the entire mint limit, preventing other users from participating in the protocol. This could be mostly mitigated by charging a small fee on mints, which is sent to the protocol's governance contract or distributed to pre-existing stakers. Could you confirm this @0xfoobar ?

0xfoobar (Alchemix) commented:

> Upon further review the griefing action of
>
> 1. initiate flashloan
> 2. deposit
> 3. mint
> 4. burn
> 5. repay flashloan
>
> would be a valid approach to grief the protocol. No funds are at risk but we'll discuss internally how to best mitigate this.

0xleastwood (judge) commented:

> Agreed, keeping this issue as is!

> This could have been upgraded to high risk if the ease of attack involved:

- Limits on burning tokens.
- The griefing attack is somewhat cheap, making it easy for attackers to maintain.

> But as the issue raised does not lead to a loss of user's funds, I believe medium risk to be justified.

[M-15] Lido adapter incorrectly calculates the price of the underlying token

Submitted by AuditsAreUS

The Lido adapter incorrectly calculates the price of WETH in terms of WstETH.

The function returns the price of WstETH in terms of stETH. The underlying token which we desire is WETH. Since stETH does not have the same value as WETH the output price incorrect.

The impact is severe as all the balance calculations require the price of the yield token converted to underlying. The incorrect price may over or understate the harvestable amount which is a core calculation in the protocol.

Proof of Concept

The function `IWstETH(token).getStETHByWstETH()` only converts WstETH to stETH. Thus, `price()` returns the value of WstETH in terms of stETH.

```
function price() external view returns (uint256) {
    return IWstETH(token).getStETHByWstETH(10**SafeERC20.exp
}
```

Recommended Mitigation Steps

Add extra steps to `price()` to approximate the rate for converting stETH to ETH. This can be done using the same curve pool that is used to convert stETH to ETH in `unwrap()`.

0xfoobar (Alchemix) disputed and commented:

> The design mechanism relies upon stETH reaching eventual 1:1 redeemability for ETH after the merge and shanghai enables withdrawals. This is core to out like-kind collateral/asset model. We will update the stETH token adapter at that time to do direct redemptions instead of Curve swaps. So in the meantime, the protocol accrues discounted assets.

0xleastwood (judge) decreased severity to Medium and commented:

> While the sponsor's comments suggest that this will be a non-issue after the merge/shanghai enables withdrawals, I believe there is legitimacy in the fact that the protocol will accrue discounted assets. It does not lead to the loss of assets, but value can be leaked if `WETH` is priced incorrectly. As such, I'm downgrading this to medium severity.

[M-16] If `totalShares` for a token falls to zero while there is `pendingCredit` the contract will become stuck

Submitted by AuditsAreUS

AlchemistV2.sol#L1290-L1300
AlchemistV2.sol#L1268
AlchemistV2.sol#L1532
AlchemistV2.sol#L899
AlchemistV2.sol#L1625

It is possible for the contract to become stuck and unable to perform any actions if the `totalShares` of a yield token fall to zero while there is some `pendingCredit` still to be paid.

It will then be impossible to call deposit or withdraw functions, mints, burns, repay, liquidate, donate or harvest due to division by zero reverts in:

- `_distributeCredit()`
- `_distributeUnlockedCredit()`
- `_calculateUnrealizedDebt()`
- `_convertSharesToYieldTokens()`
- `donate()`

Furthermore, any `pendingCredit` amount of tokens are still in the contract will become permanently stuck.

Proof of Concept

This case may arise under the follow steps a) `deposit()` is called by a user then time passes to earn some yield b) `harvest()` is called by the keeper which calls `_distributeCredit()` and increases `pendingCredit` c) `withdraw()` is called by the user to withdraw all funds

Since there is `pendingCredit` the following will have a non-zero balance for `unlockedCredit` however `yieldTokenParams.totalShares` is zero and thus we get a division by zero which reverts the entire transaction.

```
    function _distributeUnlockedCredit(address yieldToken) inter
        YieldTokenParams storage yieldTokenParams = _yieldTokens


        uint256 unlockedCredit = _calculateUnlockedCredit(yieldT
        if (unlockedCredit == 0) {
            return;
        }


        yieldTokenParams.accruedWeight     += unlockedCredit * I
        yieldTokenParams.distributedCredit += unlockedCredit;
    }
```

Each of the other listed functions will reach the same issue by attempting to divide some numerator by the `totalShares` which is zero.

Recommended Mitigation Steps

Consider preventing `totalShares` from over becoming zero once it is set. That is enforce a user to leave at least 1 unit if they are the last user to withdraw.

Another option is to transfer the first 1000 shares to a "burn" account (e.g. 0x000...01), when the first user deposits.

Alternatively, when the last user withdraws, transfer all pending credit to this user and set the required variables to zero to replicate the state before any users have deposited.

0xfoobar (Alchemix) confirmed, disagreed with severity and commented:

> Disagree with severity because given the depth of distinct users using Alchemix, it is unlikely this scenario would occur.

0xleastwood (judge) commented:

> This is an interesting issue. At the moment, it sits somewhere between medium and high risk, so I will need to think about this more before coming to a decision.

0xleastwood (judge) decreased severity to Medium and commented:

> After further thought, I think this does not fit the criteria of high severity for the following reasons:
>
> - The protocol can be DoS'd on new deployments via front-running, but it does not lead to lost funds by users. It'd only require a new deployment by the Alchemist team.
> - If the protocol was to migrate to a new version of the protocol, a mass withdrawal event could lead to locked `pendingCredit`. However, because rewards are harvested by a keeper, I believe this to be unlikely as migrations will most certainly be coordinated by the protocol and its keepers. As such, users will be aware that they would miss out on rewards if the keeper does not harvest rewards prior to the migration.
> - Rewards are regularly harvested by the keeper, and as such, the value at risk is somewhat negligible.

> For these reasons, I believe medium severity to be justified.

[M-17] Debt can be repaid with a depegged underlyingToken, which can be exploited by arbitrageurs and drives the market price of alToken to match the worst depegged underlyingToken

Submitted by WatchPug

AlchemistV2.sol#L1679-L1682

```
function _normalizeUnderlyingTokensToDebt(address underlyingToke
    return amount * _underlyingTokens[underlyingToken].conversic
}
```

AlchemistV2.sol#L743-L786

```
function repay(address underlyingToken, uint256 amount, address
    ...
    uint256 credit = _normalizeUnderlyingTokensToDebt(underlying

    // Update the recipient's debt.
    _updateDebt(recipient, -SafeCast.toInt256(credit));
    ...
```

When repaying the debt with an `underlyingToken`, the amount in terms of the `underlyingToken` (adjusted for decimals) will always be taken in a 1:1 ratio/price for the subtrahend of the debt.

We believe this design is flawed and be exploited by arbitrageurs and eventually drives the market price of alToken to match the worst depegged underlyingToken.

Because if `alToken` is trading at a higher price against the depegged underlyingToken, the arbitrageur can always mint alToken and market sell for more depegged underlyingToken and repay the debt.

Proof of Concept

Given:

- alUSD is trading at $1
- minimumCollateralization: 4/3

When USDT is slightly depegged, and trading at `$0.9`:

An arbitrageur can:

1. Deposit `100M USDC` as collateral and mint `75M alUSD` (100*0.75);
2. Market buy `75M USDT` with `67.5M alUSD` (75*0.9) and repay the debt;
3. Withdraw the `100M USDC` collateral;
4. Dump the remaining `7.5M alUSD` (75-67.5) for profit.

This can be repeated until the market price of alUSD drops to `$0.9`, the same price as USDT.

Recommendation

Consider updating the `repay()` function and change to market buy using the underlyingToken to alToken and then `burn()` the alToken to reduce debt.

[0xfoobar (Alchemix) acknowledged, disagreed with severity and commented]:

> This is a core design choice underlying the Alchemix system. Like-kind collateral and debt assumes that assets will hold their relationship. The onus lies on governance to choose safe collateral assets.

[0xleastwood (judge) decreased severity to Medium and commented]:

> I agree with what was raised by the warden but disagree with the associated severity. Because user's assets are tied to the underlying collateral, it makes sense that the a depegged token would be reflected in the price of `alToken`. As a result, arbitrageurs are free to profit off this by driving the price of `alToken` down to its true value.

> I consider this to be medium risk because of an unlikely assumption made when considering the likelihood of a depeg event. Assets are not at direct risk, but value can be leaked under certain assumptions.

Low Risk and Non-Critical Issues

For this contest, 46 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: GimelSec, AuditsAreUS, 0x1f8b, 0xsomeone, 0xDjango, joestakey, TerrierLover, cccz, fatherOfBlocks, hyh, robee, Ruhum, shenwilly, WatchPug, csanuragjain, jayjonah8, MaratCerby, BowTiedWardens, horsefacts, sikorico, tintin, catchup, cryptphi, ellahi, oyc_109, Picodes, throttle, BouSalman, 0x4non, bobirichman, Cityscape, delfin454000, mics, MiloTruck, simon135, 0xNazgul, Funen, 0xkatana, hake, Hawkeye, JC, kebabsec, kenta, samruna, *and* Waze.

Summary

Low Risk Issues

	Issue	Instances
1	Latent funds can be stolen	1
2	Low level calls don't check for contract existence	1
3	Set sane maximums for input parameters	2
4	Behavior described by comment is incomplete	1
5	Unsafe use of `transfer()` / `transferFrom()` with `IERC20`	4
6	Return values of `transfer()` / `transferFrom()` not checked	2
7	Unused/empty `receive()` function	2
8	`safeApprove()` is deprecated	32
9	Missing checks for `address(0x0)` when assigning values to `address` state variables	24

	Issue	Instances
10	`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`	1
11	Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions	3

Total: 73 instances over 11 issues

Non-critical Issues

	Issue	Instances
1	Adding a `return` statement when the function defines a named return variable, is redundant	4
2	`override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings	1
3	`public` functions not called by the contract should be declared `external` instead	12
4	`2**<n> - 1` should be re-written as `type(uint<n>).max`	1
5	`constant` s should be defined rather than using magic numbers	20
6	Redundant cast	4
7	Numeric values having to do with time should use time units for readability	1
8	Missing event for critical parameter change	3
9	Use a more recent version of solidity	12
10	Use a more recent version of solidity	1
11	Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)	2
12	Inconsistent spacing in comments	11
13	Non-library/interface files should use fixed compiler versions, not floating ones	16
14	Typos	12
15	File does not contain an SPDX Identifier	32

	Issue	Instances
16	File is missing NatSpec	27
17	NatSpec is incomplete	17
18	Event is missing `indexed` fields	111
19	Use allowlist/denylist rather than blacklist/whitelist	1

Total: 288 instances over 19 issues

[L-01] Latent funds can be stolen

If someone manages, through either a bug or a mistake (self-destructing and sending funds to the contract), another user can claim the funds as their own. Measure the balance before and after, and use the difference, rather than measuring the total balance of the contract

There is 1 instance of this issue:

```
File: contracts-full/WETHGateway.sol    #1

70            IAlchemistV2(alchemist).withdrawUnderlyingFrom(msg.
71
72            uint256 amount = WETH.balanceOf(address(this));
73            WETH.withdraw(amount);
74
75:           (bool success, ) = recipient.call{value: amount}(ne
```

https://github.com/code-423n4/2022-05-alchemix/blob/de65c34c7b6e4e94662bf508e214dcbf327984f4/contracts-full/WETHGateway.sol#L70-L75

[L-02] Low level calls don't check for contract existence

Low level calls return success if called on a destructed contract. See OpenZeppelin's Address.so which checks address.code.length

There is 1 instance of this issue:

```
File: contracts-full/libraries/TokenUtils.sol   #1

65        function safeTransfer(address token, address recipient,
66            (bool success, bytes memory data) = token.call(
67                abi.encodeWithSelector(IERC20Minimal.transfer.s
68:           );
```

https://github.com/code-423n4/2022-05-alchemix/blob/de65c34c7b6e4e94662bf508e214dcbf327984f4/contracts-full/libraries/TokenUtils.sol#L65-L68

[L-03] Set sane maximums for input parameters

There should be an upper limit to reasonable fees

There are 2 instances of this issue.

For further details on this (and the warden's other findings with multiple instances), please see their full report

[L-04] Behavior described by comment is incomplete

The event is not emitted when the fee is updated the first time (in the constructor)

There is 1 instance of this issue:

```
File: contracts-full/AlchemicTokenV2.sol   #1

50      /// @notice An event which is emitted when the flash mint
51      ///
52      /// @param fee The new flash mint fee.
53      event SetFlashMintFee(uint256 fee);
54
55      constructor(string memory _name, string memory _symbol, u
56        _setupRole(ADMIN_ROLE, msg.sender);
57        _setupRole(SENTINEL_ROLE, msg.sender);
58        _setRoleAdmin(SENTINEL_ROLE, ADMIN_ROLE);
```

```
59          _setRoleAdmin(ADMIN_ROLE, ADMIN_ROLE);
60          flashMintFee = _flashFee;
61:     }
```

https://github.com/code-423n4/2022-05-alchemix/blob/de65c34c7b6e4e94662bf508e214dcbf327984f4/contracts-full/AlchemicTokenV2.sol#L50-L61

[L-05] Unsafe use of `transfer()` / `transferFrom()` with `IERC20`

Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s `transfer()` and `transferFrom()` functions do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to `IERC20`, their function signatures do not match and therefore the calls made, revert. Use OpenZeppelin's `SafeERC20`'s `safeTransfer()` / `safeTransferFrom()` instead

There are 4 instances of this issue.

[L-06] Return values of `transfer()` / `transferFrom()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `transfer()` / `transferFrom()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually making a payment

There are 2 instances of this issue.

[L-07] Unused/empty `receive()` function

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert

There are 2 instances of this issue.

[L-08] `safeApprove()` is deprecated

Deprecated in favor of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`. If only setting the initial allowance to the value that means infinite, `safeIncreaseAllowance()` can be used instead

There are 32 instances of this issue.

[L-09] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 24 instances of this issue.

[L-10] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead.

There is 1 instance of this issue:

```
File: contracts-full/libraries/RocketPool.sol   #1

14:                 keccak256(abi.encodePacked("contract.address",
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/libraries/RocketPool.sol#L14

[L-11] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future.

There are 3 instances of this issue.

[N-01] Adding a `return` statement when the function defines a named return variable, is redundant

There are 4 instances of this issue.

[N-02] `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings

There is 1 instance of this issue:

```
File: contracts-full/AutoleverageCurveMetapool.sol   #1

20:       function _maybeConvertCurveOutput(uint256 amountOut) i
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/AutoleverageCurveMetapool.sol#L20

[N-03] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 12 instances of this issue.

[N-04] `2**<n> - 1` should be re-written as `type(uint<n>).max`

Earlier versions of solidity can use `uint<n>(-1)` instead. Expressions not including the `- 1` can often be re-written to accomodate the change (e.g. by using a `>` rather than a `>=`, which will also save some gas)

There is 1 instance of this issue:

```
File: contracts-full/libraries/SafeCast.sol   #1

13:         if (y >= 2**255) {
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/libraries/SafeCast.sol#L13

[N-05] `constant`s should be defined rather than using magic numbers

There are 20 instances of this issue.

[N-06] Redundant cast

The type of the variable is the same as the type to which the variable is being cast

There are 4 instances of this issue.

[N-07] Numeric values having to do with time should use time units for readability

There are **units** for seconds, minutes, hours, days, and weeks

There is 1 instance of this issue:

```
File: contracts-full/libraries/Limiters.sol   #1
```

```
12:       uint256 constant public MAX_COOLDOWN_BLOCKS = 7200;
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/libraries/Limiters.sol#L12

[N-08] Missing event for critical parameter change

There are 3 instances of this issue.

[N-09] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions

There are 12 instances of this issue.

[N-10] Use a more recent version of solidity

Use a solidity version of at least 0.8.4 to get `bytes.concat()` instead of `abi.encodePacked(<bytes>,<bytes>)` Use a solidity version of at least 0.8.12 to get `string.concat()` instead of `abi.encodePacked(<str>,<str>)`

There is 1 instance of this issue:

```
File: contracts-full/libraries/RocketPool.sol   #1

1:    pragma solidity >=0.5.0;
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/libraries/RocketPool.sol#L1

[N-11] Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)

There are 2 instances of this issue.

[N-12] Inconsistent spacing in comments

Some lines use `// x` and some use `//x`. The instances below point out the usages that don't follow the majority, within each file

There are 11 instances of this issue.

[N-13] Non-library/interface files should use fixed compiler versions, not floating ones

There are 16 instances of this issue.

[N-14] Typos

There are 12 instances of this issue.

[N-15] File does not contain an SPDX Identifier

There are 32 instances of this issue.

[N-16] File is missing NatSpec

There are 27 instances of this issue.

[N-17] NatSpec is incomplete

There are 17 instances of this issue.

[N-18] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

There are 111 instances of this issue.

[N-19] Use allowlist/denylist rather than blacklist/whitelist

There is 1 instance of this issue:

```
File: contracts-full/AlchemicTokenV1.sol   #1

110:   function setBlacklist(address minter) external onlySentir
```

https://github.com/code-423n4/2022-05-alchemix/blob/de65c34c7b6e4e94662bf508e214dcbf327984f4/contracts-full/AlchemicTokenV1.sol#L110

0xfoobar (Alchemix) commented:

> Incredibly comprehensive, great writeup

Gas Optimizations

For this contest, 46 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: BowTiedWardens, joestakey, 0xkatana, AlleyCat, mics, WatchPug, 0x4non, 0xNazgul, simon135, fatherOfBlocks, robee, samruna, sashik_eth, TerrierLover, _Adam, 0x1f8b, 0xf15ers, 0xsomeone, catchup, ellahi, Funen, GimelSec, MiloTruck, oyc_109, Tomio, 0xDjango, csanuragjain, hake, hansfriese, Hawkeye, ignacio, MaratCerby, sikorico, Waze, 0v3rf10w, Fitraldys, horsefacts, kenta, Randyyy, throttle, UnusualTurtle, augustg, bobirichman, Cityscape, *and* JC.

[G-01] Remove or replace unused state variables

Saves a storage slot. If the variable is assigned a non-zero value, saves Gsset (20000 gas). If it's assigned a zero value, saves Gsreset (2900 gas). If the variable remains unassigned, there is no gas savings. If the state variable is overriding an interface's public function, mark the variable as `constant` or `immutable` so that it does not use a storage slot, and manually add a getter function

For further details on this (and the warden's other gas optimizations), please see their full report.

[G-02] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot

[G-03] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (20000 gas) in the constructor, and replaces each Gwarmacces (100 gas) with a `PUSH32` (3 gas). If getters are still desired, '_' can be added to the variable name and the getter can be added manually

[G-04] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (20000 gas) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings

[G-05] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Structs have the same overhead as an array of length one

[G-06] State variables should be cached in stack variables rather than re-reading them from storage

The instances (outlined in the warden's full report) point to the second+ access of a state variable within a function. Caching will replace each Gwarmaccess (100 gas) with a much cheaper stack read. Less obvious fixes/optimizations include having

local storage variables of mappings within state variable mappings or mappings within state variable structs, having local storage variables of structs within mappings, having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

[G-07] The result of external function calls should be cached rather than re-calling the function

The instances (outlined in the warden's full report) point to the second+ call of the function within a single function

[G-08] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

[G-09] `internal` functions only called once can be inlined to save gas

Not inlining costs 20 to 40 gas because of two extra `JUMP` instructions and additional stack operations needed for function calls.

[G-10] `<array>.length` should not be looked up in every loop of a `for` -loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (100 gas)
- memory arrays use `MLOAD` (3 gas)
- calldata arrays use `CALLDATALOAD` (3 gas)

Caching the length changes each of these to a `DUP<N>` (3 gas), and gets rid of the extra `DUP<N>` needed to store the stack offset

[G-11] `++i` / `i++` should be `unchecked{++i}` / `unchecked{++i}` when it is not possible

for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas ***PER LOOP***

[G-12] `require()` / `revert()` strings longer than 32 bytes cost extra gas

[G-13] `private` functions not called by the contract should be removed to save deployment gas

[G-14] Not using the named return variables when a function returns, wastes deployment gas

[G-15] Remove unused local variable

```
File: contracts-full/TransmuterBuffer.sol   #1

515           (uint256 availableShares, uint256 lastAccruedWeigh
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/TransmuterBuffer.sol#L515

[G-16] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false

[G-17] Use a more recent version of solidity

Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath` Use a solidity version of at least 0.8.2 to get compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

[G-18] Use a more recent version of solidity

Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

```
File: contracts-full/libraries/SafeERC20.sol    #1

2   pragma solidity >=0.8.4;
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/libraries/SafeERC20.sol#L2

[G-19] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

```
File: contracts-full/interfaces/external/IProxyAdmin.sol   #1

3   pragma solidity ^0.8.0;
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/interfaces/external/IProxyAdmin.sol#L3

[G-20] It costs more gas to initialize variables to zero than to let the default of zero be applied

[G-21] `internal` functions not called by the contract should be removed to save deployment gas

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword

[G-22] `++i` costs less gas than `++i`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves 6 gas *PER LOOP*

[G-23] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

[G-24] `abi.encode()` is less efficient than `abi.encodePacked()`

```
File: contracts-full/AutoleverageBase.sol   #1
```

```
102           bytes memory params = abi.encode(Details({
103               pool: pool,
104               poolInputIndex: poolInputIndex,
105               poolOutputIndex: poolOutputIndex,
106               alchemist: alchemist,
107               yieldToken: yieldToken,
108               recipient: msg.sender,
109               targetDebt: targetDebt
110           }));
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/AutoleverageBase.sol#L102-L110

[G-25] Expressions for constant values such as a call to `keccak256()`, should use `immutable` rather than `constant`

See this issue for a detail description of the issue

[G-26] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

[G-27] Don't use `SafeMath` once the solidity version is 0.8.0 or greater

Version 0.8.0 introduces internal overflow checks, so using `SafeMath` is redundant and adds overhead

```
File: contracts-full/TransmuterBuffer.sol   #1

7   import "@openzeppelin/contracts/utils/math/SafeMath.sol";
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/TransmuterBuffer.sol#L7

[G-28] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

Saves deployment costs

```
File: contracts-full/gALCX.sol   #1

107           require(success, "Transfer failed");
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/gALCX.sol#L107

[G-29] Multiplication/division by two should use bit shifting

`<x> * 2` is equivalent to `<x> << 1` and `<x> / 2` is the same as `<x> >> 1`. The `MUL` and `DIV` opcodes cost 5 gas, whereas `SHL` and `SHR` only cost 3 gas

```
File: contracts-full/ThreePoolAssetManager.sol   #1

355               if ((examineBalance = (v.maximum + v.minimum)
```

https://github.com/code-423n4/2022-05-alchemix/blob/71abbe683dfd5c0686b7e594fb4f78a14b668d8b/contracts-full/ThreePoolAssetManager.sol#L355

[G-30] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement,

because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`)

[G-31] Use custom errors rather than `revert()` / `require()` strings to save deployment gas

Custom errors are available from solidity version 0.8.4. The instances below match or exceed that version

[G-32] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost

[G-33] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public` and can save gas by doing so.

0xfoobar (Alchemix) commented:

> Incredibly comprehensive, great work with the explanations and detailed line number links

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top