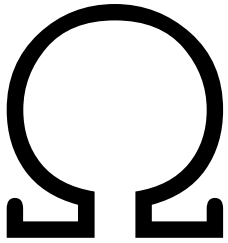
Altitude

Final Audit Report

First version: August 1, 2022
Updated in September 6, 2023
Updated on March 21, 2024



Team Omega

teamomega.eth.limo

Summary

Altitude has asked Team Omega to audit their smart contract system. The following document was mostly written between July and October 2022, with a final update on September 6, 2023.

We found **9 high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **29** issues as "medium" - these are issues we believe you should definitely address, even if they do not lead to loss of funds. In addition, **45** issues were classified as "low", and **16** issues were classified as "info" - we believe the code would improve if these issues were addressed as well.

Many, but not all, of these issues were resolved in later commits. Please see the "scope" section for details.

Severity	Number of issues	Number of resolved issues
High	9	9
Medium	19	18
Low	46	39
Info	22	16

Scope of the Audit

We audited the code from the following repository:

`https://github.com/refi-network/protocol-v1-audit/`

History of this document

This document reflects the issues found in our audit of the Altitude Code in October 2022. The audit took place over two “snapshots”.

- We have audited the Solidity code at commit `ee94c29c5e5c6bef58f6de65dfcb8eb3a74db288` found under the following directories (located under the `contracts` directory): `common`, `decision-makers`, `libraries`, `misc`, `tokens`, and `vaults`. The code under the `libraries/uniswap-v3` directory and in the `libraries/RMath.sol` file was not audited.
- For the commit `8a79b2014d004ea595839b5fa0759df2d921ef88` we have audited the Solidity code found under the `oracles` and `strategies` directories (located under the `contracts` directory)

The issues that we found were subsequently addresses in a series of commits:

- A number of issues were addressed in commit `25dd16ccd13164f9c2b7ba63743bd07f049813d2` on October 27, 2022. This commit also

contains some other changes to the logic. We have audited the changes with respect to commit `8a79`, and updated the issues below.

- On September 5, 2022, A new version of the contracts was created at commit `f19e2e5edeb87e5e37345932cede3a0e99709f23`. A full audit of the changes is the subject of a separate audit.
- On March 16, 2024, some remaining issues were resolved in commit `37b5b6e20651d3e876630686764ec3a5443dc1fb`. We updated the report to reflect those changes

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc

Info	Matters of opinion

Findings

General

G1. Avoid passing asset addresses to strategies [low] [resolved]

In its current architecture, each strategy contract is meant to be used for a single vault, and each vault has a single pair of assets it works with. These assets are constant for the vault. Yet the interfaces and implementations of all strategies accept as inputs to their functions the assets they should work with. This is unnecessary and confusing, since the strategy should always work with the same supply and borrow assets, and passing a different address in some cases leads to unexpected behavior (SC2 is an example).
In addition, as shown in issues SA1 and SA3, this ambiguous behavior can lead to programming errors.
Recommendation: Save the supply and borrow assets in immutable state variables in the constructors of the lending and farming strategies, and avoid receiving them as parameters in functions.
Severity: Low.
Resolution: The issue was resolved. The supply and borrow assets are set in the `setVault` function, and used instead of passing them for each function call.

G2. Licensing and copyright issues [medium] [resolved]

There are several problems with the licensing of the code in the repository.
The only licensing information we could find in the repository are the `SPDX-License-Identifier` in the solidity files, and the `license` field in `package.json`. These SPDX identifiers denote a number of different licenses: `BUSL-1.1`, `agpl-3.0`, `AGPL-3.0`, `MIT`, `GPL-2.0-or-later`, and, in `package.json`, `ISC`.
Each of these licenses require that you include a copy of the text of the license, and a short text in which you claim the copyright (claiming authorship of the code is a necessary condition for granting any further rights to the code).
In addition, there are a number of library files that are taken from other projects (such as the files in the uniswap library). These files are originally released under licensing conditions which you are not respecting: specifically the requirement that you include the full text of the original license and the original copyright claim. You must also check if these licensing are compatible with the license under which you intend to publish the overall package (presumably `ISC`).

Recommendation: Respect the conditions of the licenses of third-party software that you include. Have a more consistent licensing policy (and use only valid SPDX identifiers) - if possible decide on a single license for the software that you have written. Include a copyright claim for the code that you have written. Include the `LICENSE.txt` or similar in which the copyright and licensing conditions are made explicit.
Severity: Medium
Resolution: Licensing information is now consistent throughout the code base

G3. Use the latest Solidity version [low] [resolved]

The latest release of the Solidity compiler is 0.8.15. It contains some bug fixes relative to the version that is used in the code base we reviewed (0.8.14).
Recommendation: Although none of the bug fixes in 0.8.15 seem directly relevant to the code base under review, we recommend in any case to use the latest (minor) version.
Severity: Low
Resolution: The issue was resolved as recommended.

G4. Pin dependencies on solidity packages [low] [resolved]

The `package.json` file specifies a number of dependencies of solidity code:

```
"devDependencies": {
    …
    "@openzeppelin/contracts": "4.4.2",
    "@openzeppelin/contracts-upgradeable": "4.5.1",
    "@openzeppelin/upgrades": "2.8.0",
    …
    "@uniswap/v2-core": "^1.0.1",
    "@uniswap/v2-periphery": "^1.1.0-beta.0",
    …
},
"dependencies": {
    "@chainlink/contracts": "^0.4.1",
    "@uniswap/v3-core": "^1.0.1",
    "@uniswap/v3-periphery": "^1.4.0",
    …
 }
```

The code from some of these packages is imported in the solidity files, and is part of the compiled result that will be uploaded on-chain. Using fixed dependencies is important to get a predictable result from the compilation, and will make future verification of the contracts easier.

Recommendation: Remove all dependencies of solidity packages that are not used from the package.json (i.e. `@openzeppelin/contracts-upgradeable`, `@openzeppelin/upgrades`, `@uniswap/v2-core`, `@uniswap/v2-periphery`. Of the remaining packages, pin the version that you intend to use (currently, on the basis of `package-lock.json`, this would be:

```
"@chainlink/contracts": "0.4.1",
"@uniswap/v3-core": "1.0.1",
"@uniswap/v3-periphery": "1.4.1",
```

Also, move the `@openzeppelin/contracts` dependency from the `devDependencies` to the regular `dependencies` section.
Severity: Low
Resolution: The issue was resolved as recommended.

G5. Use Error objects instead of error strings [low] [partially resolved]

Solidity 0.8.4 introduced custom errors. These new error objects are used in some, but not all, parts of the code. In many places, the code still uses custom error strings and, in some places, such as in `GroomableManager.sol`, it uses an error string for which a custom error has been defined:

```
require(!safetyMode, "OM_V1_NO_PRICE_DISCREPANCY");
```

Recommendation: Use custom error objects everywhere for consistency, and to save some gas.
Severity: Low
Resolution: The issue was partially resolved. Error strings were replaced for error objects in some places across the codebase, but not in all of it.

G6. Declare variables immutable where appropriate [low] [resolved]

Variables that are set in the constructor but then are not changed after can be declared `immutable` to save some gas. Examples are:

```
MigrationDecisionMaker.decisionValidatorsCount
VaultRegistryV1.WETH
VaultRegistryV1.vaultFactory
VaultRegistryV1.vaultETHFactory
VaultRegistryV1.tokensFactory
InterestToken.vault
InterestToken.underlying
InterestToken._decimals
```

```
SafetyModeDecisionMaker.discrepancyThreshold
etc...
```

Recommendation: Declare variables that are only set in the constructor `immutable` for more clarity, and to save some gas.
Severity: Low
Resolution: The issue was resolved as recommended.

-G7. Incomplete test coverage [low] [not resolved]

The tests do not cover the entire code base. A part of the un-tested code contains error conditions, but there are also missing tests for other important scenarios, in which for example a harvest, or user gains, are negative. Also the scenario of storing partial commits is not tested. Some of the issues below point to these and other un-tested scenarios.
The appendix contains the output of the coverage report, but note that even with coverage at 100%, there may still be meaningful scenarios that remain untested.
Recommendation: Try to have a test suite that completely covers the code base.
Severity: Low
Resolution: The issue was not resolved. Some tests are failing. Also see G9 on CI issues.

G8. Comparatively high gas costs [info] [resolved]

Below are Altitude's gas costs on some of the VaultERC20 interactions (average gas costs as reported by hardhat based on the test suite):

Altitude average gas costs	commit `ee94`
Borrow	823,339
Deposit	723,653
Repay	619,215
Withdraw	625,768

These costs are relatively high, compared with the gas costs of some other known systems (data from https://crypto.com/defi/dashboard/gas-fees).

Provide liquidity	
Uniswap V2	593,954
Balancer	285,031
SushiSwap	258,443
Curve	170,357
Mooniswap	161,400

Remove liquidity	
SushiSwap	200,836
Uniswap V2	191,667
Mooniswap	149,002
Balancer	130,479
Curve	120,762

We have included in the report some potential gas optimizations you could implement to reduce these costs, but also more generally, we recommend that as you work on the fixes of the issues, you should also try to follow more closely best practices for gas saving, such as reading from memory instead of storage when possible and avoiding unnecessary saving of data to storage when events could be used instead.

Severity: Info

Resolution: The issue was resolved. Average costs for the listed actions are now lower by about 40% to 70%. These are the average gas costs for commit `25dd`

Altitude average gas costs	commit `ee94`	commit `25dd`
Borrow	823,339	236,381
Deposit	723,653	406,136
Repay	619,215	355,002
Withdraw	625,768	350,504

G9. Continuous Integration fails [info] [new]

The Continuous Integration (CI) of the project fails, but its results are ignored.
Recommendation: Fix CI, and require that the CI passes on new PRs to be able to merge them.
Severity: Info
Resolution: The issue was not resolved

RolesManageable.sol

RM1. onlyRole modifier by default does not limit access [low] [resolved]

The `onlyRole` modifier is meant to limit access to certain functions so that they can be called only by accounts with the corresponding role assigned.
However, by default, if the `functionsRoles` mapping is not set for a certain function signature, the corresponding function will be callable by any address that does *not* have a role set.
This is counterintuitive, and it makes it easy to make mistakes.
Recommendation: Make `onlyRole` revert if the `functionsRoles[fn]` is empty. Or remove the `RolesManageable` contract altogether and instead use the OpenZeppelin `AccessControl` contract, which is a more standard and well-tested option for the same use case.
Severity: Low
Resolution: The issue was resolved as recommended. The `onlyRole` modifier will now revert if `functionsRoles[fn]` is empty.

MigrationDecisionMaker.sol

MD1. Check for duplicates when setting decisionValidators in constructor [low] [resolved]

In the constructor, the list of valid `decisionValidators` is set. There are no checks for duplicates. Without such a check, it is not clear how many decision validators will actually be set, and so the `validMajorityCount` check is meaningless.
Recommendation: Check for duplicates when setting the list of `decisionValidators`. Also consider checking that the zero address is not included in the list.
Severity: Low
Resolution: The issue was resolved as recommended.

MD2. Use EIP-712 for signing messages [low] [resolved]

The message that decisionValidators are to sign to approve a migration is constructed as follows:

```
abi.encode(newStrategyAddress, nonce, address(this))
```

There are standards for signing off-chain messages. Following these standards makes integration with wallets easier (as they know how to parse the messages), and also avoids known pitfalls. For example, in your case, there is a theoretical possibility that signatures are re-used on other chains - which is why EIP-712 also includes the chainId of the message that is to be signed.
Recommendation: Use the EIP-712 standard to construct messages:
https://eips.ethereum.org/EIPS/eip-712
Severity: Low
Resolution: The issue was resolved as recommended.

RebalanceDecisionMaker.sol

RD1. Remove unused state variables [low] [resolved]

The state variables `emergencyMinTargetHealthFactor` and `emergencyMaxTargetHealthFactor` are not used and can be removed.
Recommendation: Remove the `emergencyMinTargetHealthFactor` and `emergencyMaxTargetHealthFactor` variables and the related code.
Severity: Low
Resolution: The issue was resolved. The RebalanceDecisionMaker.sol file was removed and replaced with the RebalanceIncentivesController.sol file, which does not contain those variables.

SafetyDecisionMaker.sol (FarmModeDecisionMaker.sol)

This file was renamed from `SafetyDecisionMaker.sol` to `FarmModeDecisionMaker.sol`

SD1. Linked list initialization does not check for valid input [low] [resolved]

This contract uses a custom linked list implementation to store a list of oracles in a `priceSources` mapping. The linked list is initialized in the constructor, and can be expanded using `setPriceSources`.

The constructor does not do basic checks for soundness of the input data - for example, if the list contains the `SENTINEL_PRICE_SOURCE` value, or any duplicates, the list will not behave as expected. The effect can be that `checkPriceDiscrepancy` will report a price discrepancy even if more than `sourcesThreshold` oracles do confirm the price. Also `setPriceSources` may produce unpredictable results if the list is corrupted in this way.
Recommendation: Check for duplicates and the presence of `SENTINAL_PRICE_SOURCE` in the constructor. Or abandon the idea of defining custom data structures altogether, and use a simple array for storing and accessing the list of sources.
Severity: Low
Resolution: The issue was resolved as recommended. The `SafetyDecisionMaker.sol` file was removed and replaced with the `FarmModeDecisionMaker.sol` file, in which the use of linked list has been abandoned in favor of a simple array. However, the `getPriceSources` function is now overly complicated, and can just return the `priceSources` array directly instead of copying it first. Or, instead, the `priceSources` array could be made public, and the `getPriceSources` function removed.

SD2. Remove inactive sources from list [low] [resolved]

The `setPriceSources` function handles adding sources to the list, and activating or deactivating them. The way that this method is implemented is such that the list will only grow. This adds unnecessary gas costs each time the decision maker is checked, and in extreme cases may cause the function to run out of gas. In these cases, there is no other option but to replace the decision maker in the vault with a new instance.
Recommendation: Remove items from the list, instead of setting active to false.
Severity: Low
Resolution: The issue was resolved as recommended. The `SafetyDecisionMaker.sol` file was removed and replaced with the `FarmModeDecisionMaker.sol` file, in which price sources can be added to the list or removed from it, instead of having an option to activate and deactivate them.

SD3. No guarantee that sourcesThreshold < number of price sources [low] [resolved]

The function `setPriceSources` contains the following lines of code.

```
toSet
          ? (newSourcesThreshold < sourcesThreshold)
          : (newSourcesThreshold > sourcesThreshold)
  ) {
      revert Errors.OM_INVALID_SOURCE_THRESHOLD();
  }
```

The intention seems to be to guarantee that `sourcesThreshold` stays below the number of sources (because in that case, the `checkPriceDiscrepancy` function would always return true (and the vault would be in permanent safety mode). Yet this check does not guarantee this. At the same time, there are valid use cases in which it is expedient to reduce the number of price sources but raise the threshold - something that will now have to be done in two separate transactions.
Recommendation: Simply check that the `newSourcesThreshold <= priceSourcesCount`.
Severity: Low
Resolution: The issue was resolved as recommended. The SafetyDecisionMaker.sol file was removed and replaced with the FarmModeDecisionMaker.sol file, in which, when adding or removing price sources, there is a check that `newSourcesThreshold <= priceSourcesCount`.

HarvestHelper.sol

HH1. Harvesting while vault active assets are negative reverses the yield distribution curve [high] [resolved]

When calling `harvest`, if the value `vaultActiveAssets` is negative (which may happen if a large amount of tokens were borrowed by users), the distribution curve of the farm earning is reversed, so that instead of the user who contribute to the harvest receiving its earnings, the user with negative active assets will be the ones rewarded.

For example, suppose there are `farmEarnings` worth $1000 to distribute, the amount of `vaultActiveAssets` is -1, and the user has -1000 tokens of active assets (i.e. is 1000 tokens in debt above the target threshold). In that case, this user who has a negative active assets would be allocated $1,000,000 worth of `farmEarnings`, meaning the user who burdens the system received both the reward and payment from the users with whose funds the reward was generated.
Severity: High
Recommendation: Disallow harvesting while the `vaultActiveAssets` are negative.
Resolution: The issue was resolved. It is no longer possible to harvest while the `vaultActiveAssets` is negative.

HH1b. Unfair allocation of harvest earnings [medium] [partially resolved]

In `userCalculateCommit`, the earnings from farming are allocated to the user on the basis of its share of “active assets” relative to the “active assets” of the vault:

```
userHarvestChange = userActiveAssets
      .absMul(currentHarvest.farmEarnings)
```

```
            .absDiv(currentHarvest.vaultActiveAssets)
```

Roughly, the “active assets” of an account are calculated as the value of the assets the account contributed multiplied by `targetThreshold`, minus the value of the assets she has borrowed.

The value of `userActiveAssets` can be negative (as the relative price of supply and borrow tokens changes over time, and in any case the amount that a user can borrow is determined by `supplyThreshold` and not `targetThreshold`, which is used to calculate the active assets).

The value `vaultActiveAssets` can be arbitrarily small, which may generate extreme situations.

For example, suppose `vaultActiveAssets` is 1 wei (i.e. almost 0), and the active assets of Alice are 1e9 we (typically 10 tokens), and the `farmEarnings` are worth $1. These values are small but not unusual. In this case, Alice would be entitled to 1 billion dollars, which are to be paid by other users.

Recommendation: “Share of active assets” is not a correct way to divide up the farm earnings. One possible adaption of the algorithm to use “supplied minus borrowed” (instead of “threshold * supplied - borrowed”).
Severity: High
Resolution: The issue was partially resolved. The algorithm now uses the user’s liquidation threshold for the calculation of active assets instead of the target threshold, which is typically much higher. This reduces the chance of such edge cases as described, as these positions would be liquidated before the liquidation threshold is reached.

However, in more extreme situations, some of the detailed scenarios may still occur. For example, in

```
        userHarvestChange = userActiveAssets
                    .absMul(currentHarvest.farmEarnings)
                    .absDiv(currentHarvest.vaultActiveAssets.value);
        }
```

If `vaultActiveAssets` is much smaller than `userActiveAssets` (i.e. the vault is close to liquidation, but the user has a healthy balance) the `userHarvestChange` could be much higher than the `farmEarnings` that are to be distributed.

The updated algorithm also introduces new cases that seem unfair or unexpected. For example if the target threshold (and so also the `divertEarningsThreshold`) is set to 50% and the liquidation threshold is 80%, someone borrowing at 49% will receive the 31% reward. But someone borrowing at 51% will receive no reward at all.

HH2. Unnecessary constant addition to user harvest [low] [resolved]

In line 102 the calculation of the `userHarvestChangeNew` adds `1e18/2` to the calculation.
This has just a minimal effect on the result, but has no reason to be there and should be removed.
Recommendation: Remove the addition of `1e18/2` in line 102.
Severity: Low
Resolution: The issue was resolved as recommended.

GroomableManager.sol

GM1. Add maximum migration fee [low] [resolved]

When migrating the lending strategy, the contract will use a flash loan to perform the migration of funds, paying a "migration fee" to the flash loan lender. Since the fee for the loan comes from a 3rd party contract, it is not capped in any way, and could be as high as the flash loan lending strategy decides. In extreme cases, this could pose a risk of abuse by the strategy used for the flash loan to potentially take any amount, but also, it could simply be that the amount taken is higher than what the migration was expected to cost.
Recommendation: Add a maximum migration fee parameter to the vault or the migration call to ensure the fee paid for the loan is within reasonable limits.
Severity: Low
Resolution: The issue was resolved as recommended. The vault now has a max migration fee percentage that limits the fee that can be paid on migration.

GM2. Reconsider canRebalance logic [low] [resolved]

The function `canRebalance` works as a limitation on the conditions under which a vault can be rebalanced (i.e. it controls in which states the vault's borrow exposure to be closer to the desired `targetThreshold`). It does this by checking if the lender strategy's "health factor" is outside of the limits set by `minTargetHealthFactor` and `maxTargetHealthFactor` - if the reported health factor is within these limits, rebalancing will not occur.

The target borrow rate of the vault is set by `targetThreshold`, and is not related to the liquidation threshold in the lender strategy (although typically it would be set to a value well below that liquidation threshold). This logic is problematic, because whether or not the current position's health factor is between some fixed limits has no relevance for disallowing a call to `rebalance` - i.e. a perfectly healthy position may need rebalancing, just as a position that is very close to liquidation.

Recommendation: Remove the current `canRebalance` check from the `rebalance` function, as it does not seem to do anything useful at all, and can hinder healthier vault operation, especially if `minTargetHealthFactor` and `maxTargetHealthFactor` denote a range close to the liquidation threshold of the lender.
Severity: Low
Resolution: The issue was resolved as recommended. It is now possible to call `rebalance` at any point.

HarvestableManager.sol

HM1. User harvest and vault reserves are counted twice when calling storeCommitCalculation [high] [resolved]

When calling `storeCommitCalculation`, it will call the `HarvestHelper.userCalculateCommit`, then add its result to the current `commit.userHarvestUncommitted` and `commit.vaultReserveUncommitted`. Yet the `HarvestHelper.userCalculateCommit` already returns the `userHarvestUncommitted` and `vaultReserveUncommitted` as the original values plus the changes, which means these additions of the results to the original values are actually counting the original values twice. The implications of this are that when a user is being committed using the `storeCommitCalculation`, their `userHarvestUncommitted` and `vaultReserveUncommitted` values will be too big or too small than they should be, and the entire calculation will become incorrect. Also, this may cause the subtraction on line 299 to underflow, as the user harvest could potentially grow bigger than the entire harvest. In that case not only will the user harvest calculation be wrong, but the user will also be unable to interact with any functions that call the `commitUser`, including `deposit`, `borrow`, `withdraw`, and `repay`. Essentially leaving the user completely unable to interact with the system.
Recommendation: Instead of adding to their current values, assign the result returned from the `HarvestHelper.userCalculateCommit` directly to the `commit.userHarvestUncommitted` and `commit.vaultReserveUncommitted`, just like done for the `commit.harvestIndex`. We also suggest adding tests for the `storeCommitCalculation`, as it is currently only tested as called from the `commitUser`, but should also be tested when called independently and when used for partial harvest committing.
Severity: High
Resolution: The issue was resolved as recommended.

HM2. Users may lose parts of their harvest earnings in griefing attacks [medium] [resolved]

Each harvest is allocated to each account on the basis of the proportion of capital that the account has provided during the harvest period.

A pseudo-formula expressing this could be:

```
userHarvestShare =
      userActiveAssetsAtHarvestTime/totalActiveAssetsAtHarvestTime
      * userAssetsProvidedPeriod/harvestPeriod
      * totalHarvest
```

This algorithm may cause users to lose part of their harvest earnings when interacting with the vault. We are analyzing here what happens on `transfer`, but similar logic is implemented for `transferFrom`, `deposit` and `repay`.

If a user receives capital because another user transfers supply tokens to her account, her assets from before the deposit action are not counted when calculating her share of the harvest:

```
userHarvestShare =
      userActiveAssetsAtHarvestTime/totalActiveAssetsAtHarvestTime
      * timeBetweenBalanceChangeAndHarvestTime/harvestPeriod
      * totalHarvest
```

I.e. she will get her share of the harvest proportional only from the moment she got these extra tokens. If she already has 100 tokens deposited, and then another user sends her 1 additional token in the last block before the harvest (or if she deposits this token herself), and the harvest period was 100 blocks, she will only get about 1% of the rewards that she would have gotten if she had not deposited.

This does not only seem unfair, it also offers malicious users a path to a *griefing attack*, in which depositors can be cheated out of the earnings of a harvest period by sending the micro-amounts of tokens just before harvest time. Also the `repay` function can be used for the same kind of attack by repaying a 0 (see also issue C6) or infinitesimal amount “on behalf of” a user.

Recommendation: Consider an approach in which earnings are allocated on the basis of capital provided during the entire harvest period (instead of a snapshot at the end of the harvest period). This could be done by keeping track, for each user and the vault as a whole, of the amount of “token-block” provided during a harvest period, which would be calculated as:

```
amountOfTokensProvided * amountOfBlocksDuringWhichTheseTokensWereProvided
```

Severity: Medium
Resolution: This issue was resolved. For now, the functions that may affect the earning of a user from the harvest can only be called by accounts approved by the user (as long as `allowlistActive` is true), which will prevent a griefing attack. Note that users may still suffer losses of harvest reward when depositing, transferring, or repaying (intentionally by their own action or any address they approved to do so), but since this can only happen as part of a conscious action of a user, she can take into account the potential loss of harvest reward before deciding whatever to take the action or not.

HM2b. Users may lose harvest earnings when others deposit [low] [not resolved]

Each harvest is allocated to each account on the basis of the proportion of capital that the account has provided during the harvest period.

A pseudo-formula expressing this could be:

```
userHarvestShare =
      userActiveAssetsAtHarvestTime/totalActiveAssetsAtHarvestTime
      * userAssetsProvidedPeriod/harvestPeriod
      * totalHarvest
```

An undesirable effect of this accounting mechanism is that users whose balance does not change at all during the harvest period can be negatively affected by actions from other users, i.e. a user providing capital but not interacting in any other way with the protocol will be penalized if other users join. For a concrete example, suppose the `harvestPeriod` lasts 1000 blocks and pays out 1000 tokens. User A provides 100% of the capital during the 999 blocks, and user B makes a deposit doubling the supply in the 999th block. At harvest time, user A has provided only half of the total capital, and will get only 500 tokens from the harvest, even though her capital has generated approximately 99% of the harvest gains.

Recommendation: Consider an approach in which earnings are allocated on the basis of capital provided during the entire harvest period (instead of a snapshot at the end of the harvest period). This could be done by keeping track, for each user and the vault as a whole, of the amount of “token-block” provided during a harvest period, which would be calculated as

```
amountOfTokensProvided * amountOfBlocksDuringWhichTheseTokensWereProvided
```

Severity: Low
Resolution: The project has acknowledged the issue.

HM3. Missing error check on recognise farm rewards when calling harvest [low] [resolved]

When calling `harvest` there's a call to recognise the farm rewards (lines 65 - 66), which might return an error code if it fails. Yet there is no check to ensure that the call was successful, like there is for example for recognizing lender rewards (line 74). This means that the harvest will still go through even if the farming strategy has returned an error.
Recommendation: Add the same require statement that exists in line 74 also right after recognizing the farm rewards (between line 66 and 67).
Severity: Low
Resolution: The issue was resolved. The harvest function will now revert if either one of the calls to recognize rewards fail.

HM4. Read total earnings from memory instead of recalculating it [low] [resolved]

In the `harvest` function, lines 106 to 108, the farm earnings and lender rewards of the harvest are added to the `harvestStorage.harvestEarningsUncommitted`, but it could save gas to just add the `totalEarnings` (declared in line 91), which already holds the sum of these two numbers in memory, and so could save gas to use it instead of re-summing those numbers.
Recommendation: Change lines 106 - 108 to: `harvestStorage.harvestEarningsUncommitted += totalEarnings;`
Severity: Low
Resolution: The issue was resolved. Both variables are no longer present in the code, and the new variables that are used are read from memory when needed.

HM5. Remove unused state variables [low] [partially resolved]

The `UserHarvestData` that is saved for each user contains multiple variables that are written to, but never read from. These include the `supplyIndex`, `borrowIndex`, `harvestEarnings`, and `harvestCosts`. These variables could just be emitted in events instead of saved to storage, or even entirely removed to save gas.
Recommendation: Remove the `UserHarvestData´s` `supplyIndex`, `borrowIndex`, `harvestEarnings`, and `harvestCosts` variables, and emit their values in events if needed.
Severity: Low
Resolution: The issue was partially resolved. The `supplyIndex` and `borrowIndex` were removed, but the `harvestEarnings`, and `harvestCosts` are still present.

HM6. Bad incentives structure in commitUsers [low] [resolved]

The `commitUsers` function will call the `IncentivesManager.sendRewards` to reward callers who do commits for multiple users. The `commitUsers` will pass the number of harvests committed as a multiplier for the reward, and in theory (as no reward logic exists yet in the `IncentivesManager.sol`), the reward will then use this multiplier to calculate how much to reward the caller, compensating for their gas costs and maybe adding an extra reward. This however ignores the fact that every commit might have a different gas cost for the caller, and so a linear multiplier might not make sense. It also ignores the possibility where a user only commits themselves, just to take advantage of the reward and so making later interactions with the contract cheaper as the `commitUser` will no longer take as much gas when they interact with the vault.
Recommendation: Instead of using the harvests committed as a multiplier, pass the actual gas costs for each commit that has committed at least one harvest. About the possible user self redeeming with this, there is no straightforward and complete solution, but to at least improve the situation, it is possible to require the `commitUsers` to be called with more than one address, or only on addresses that have been inactive for many harvests.
Severity: Low
Resolution: The issue was resolved. The `IncentivesManager.sol` was entirely removed from the code.

HarvestableVault.sol

HV1a. Owner can withdraw unclaimed user rewards [medium] [resolved]

The `withdrawReserve` function allows the owner to withdraw up to `vaultReserve` tokens from the farming strategy.

Operations such as `rebalance` (especially if the `targetThreshold` is low or equal to 0) and `_repayVaultDebt` move funds out of the farming strategy without reserving funds to pay out user rewards or the vault reserve. These functions will pay off the vault's debt, and will leave any remaining `borrowUnderlying` tokens in the vault's balance.

Closing the farming position in this way has several adverse effects.

First of all, some or all of the funds which are earmarked for paying out rewards will be claimable by the owner using `withdrawReserve`, which allows the owner to take all excess `borrowUnderlying` tokens from the vault's balance.

Secondly, the owner will be able to claim `vaultReserve` amount of tokens several times - i.e. suppose the farming strategy has at least `vaultReserve` amount of tokens more than it needs to repay to the lending strategy (which would be typical in normal conditions). The owner can close the farming position, call `withdrawReserve` and take at least `vaultReserve` amount of tokens from the surplus balance of the vault, then rebalance and open a new farming position, and then withdraw `vaultReserve` tokens from the farming strategy (leaving some bad debt).
Recommendation: Respect the values of `userClaimableEarnings` and `vaultReserve` when rebalancing or repaying the vault debt. Or, alternatively (but this would require a more extensive revision), rewrite the `withdrawReserve` and `claimEarnings` functions so they will not be affected by rebalancing.
Resolution: The issue was resolved. The owner can no longer take more than the reserved amount.

HV1b. Harvested tokens claimable by as rewards are not reserved for users [medium] [not resolved]

When claiming rewards using `claimRewards`, the user's rewards are withdrawn from the farming strategy to be sent to the user. Operations such as `rebalance` (especially if the `targetThreshold` is low or equal to 0) and `_repayVaultDebt` move funds out of the farming strategy without reserving funds to pay out user rewards or the vault reserve. These functions will pay off the vault's debt, and will leave any remaining `borrowUnderlying` tokens in the vault's balance. This means that when closing the farm's position, users will not be able to claim their rewards anymore - `claimRewards` will revert.
Recommendation: Respect the values of `userClaimableEarnings` and `vaultReserve` when rebalancing or repaying the vault debt. Or, alternatively (but this would require a more extensive revision), rewrite the `withdrawReserve` and `claimEarnings` functions so they will not be affected by rebalancing.
Resolution: The issue was not resolved.

InterestToken.sol

IT1a. All user balances are zero when vault is liquidated entirely [medium] [resolved]

This issue concerns the case in which the vault's position in the lender strategy is liquidated entirely - i.e. when all of the debt in the lending strategy is paid off with supply tokens deposited in the strategy.

The function `calcNewIndex()` is responsible for calculating the new values for the `interestIndex` and `balanceAtIndex`. Because the debt was liquidated entirely, `currentLenderBalance` returns a value of 0 for the new balance. Suppose also for the sake of argument that `bufferRefill` is 0. In that case, `balanceNew == 0`, and the calculation on line 310 will set `newInterestIndex` to 0:

```
uint256 newInterestIndex = interestIndex_ -
      (interestIndex_ * (balancePrev - balanceNew) / balancePrev));
```

Setting `interestIndex` to 0 has the practical effect that any future successful calls of `calcNewIndex()` will set `interestIndex` and `balanceAtIndex` to 0 (i.e. the value will never recover).
Recommendation: Never set `interestIndex` to 0, instead, set it back to its initial value of `MATH_UNITS`.
Severity: Medium
Resolution: The issue was resolved as recommended.

IT1b. The proceeds from liquidation are not redistributed [medium] [resolved]

This issue concerns the case in which the vault's position in the lender strategy is liquidated - i.e. when a part or all of the debt in the lending strategy is paid off with supply tokens deposited in the strategy.

There might still be tokens in the farming contract that were borrowed by the vault itself, but there is no clear way to re-distribute these tokens among the depositors of the protocol.

Recommendation: If there are still tokens left in the farming, they should be distributed in a form that will be fair for the depositors.
Severity: High
Resolution: The issue was resolved. In case the vault is liquidated, the harvest will now distribute the tokens borrowed by the vault as part of the harvest gains. Also, a harvest will now happen automatically as part of the next action after such liquidation of the vault.

IT1c. Vault liquidation redistributes supply tokens from depositors to borrowers" [medium] [resolved]

This issue concerns the case in which the vault's position in the lender strategy is liquidated - i.e. when a part or all of the debt in the lending strategy is paid off with supply tokens deposited in the strategy.

When (part of) the vault’s debt is liquidated, the vault’s collateral of supply tokens is exchanged for borrow tokens that are then used to pay off the vault’s debt. This operation is then reflected on the users’ position as follows:

1. The user’s balance of deposit tokens is diminished, proportional for each user

2. Both the debt balance of the user and of the vault are diminished, proportional to the debt position of the user
3. Any surplus of debt tokens that remain in the farming strategy after liquidation are redistributed over the users, proportional to the "active assets" of the user (users get more if they have a low LTV, and less if they have a high LTV)

These are basically three separate ways of allocating the liquidation costs to users of the system:

1. The costs of the liquidation (i.e. the collateral used for paying off the loan) is distributed over all depositors, independently of whether they borrowed themselves or not
2. The proceeds of the liquidation (i.e. the part of the loan paid for by the collateral provided in the previous step) is distributed over borrowers of the system (i.e. lenders that do not borrow do not participate in this step)
3. The part of vault's debt that was paid off (in step 2) is redistributed over all users according to their active assets

The problem is that in step (1) the collateral of *all* users is used to pay off the debt, while in step (2), only borrowers get a share of the proceeds. This means that there is a transfer of value (which can be quite consistent) from non-borrowers to borrowers. This is partly compensated for by redistributing the vault's paid off debt over all users in step (3). But if the vault's loan is 0, or comparatively small, this third redistribution step will not be enough to compensate all users.

To see that this is unfair, consider the following example:

Start Values	eth price=	$2						
	Deposit, ETH	**Deposit Value**	**Debt (2 protocol**	**Cash**	**net worth**			**Lic**
Alice	100	$200.00	$0.00	$0.00	$200.00			
Bob	100	$200.00	$50.00	$50.00	$200.00			
			Debt (to lender)	***farming***				
Vault	*200*	*$400.00*	*$50.00*	$0.00	*$0.00*			
		farm=						
Pre Liquidation	eth price =	$1.00						
	Deposit, ETH	**Deposit Value**	**Debt (2 protocol**	**Cash**	**net worth**			**Lic**
Alice	100	$100.00	$0.00	$0.00	$100.00			
Bob	100	$100.00	$50.00	$50.00	$100.00			
			Debt (to lender)	***farming***				
Vault	*200*	*$200.00*	*$50.00*	$0.00	*$0.00*			
Post Liquidation	eth price=	$1.00	***SOLD 50 ETH FOR 50$***					
	Deposit, ETH	**Deposit Value**	**Debt (2 protocol**	**Cash**	**Adjustments***	**net worth**	**sition gain/loss**	
Alice	75.00	$75.00	$0.00	$0.00	$0.00	$75.00	-25.0%	
Bob	75.00	$75.00	$0	$50.00	$0.00	$125.00	25.0%	
			Debt (to lender)	***farming***				
Vault	*150.00*	*$150.00*	*$0*	$0.00		*$0.00*		

Suppose the price of the deposit token is $2, and that of the borrow token is $1. Alice deposits 200$ worth of supply tokens, and so does Bob, who also borrows 50$ of borrow tokens. Suppose no farming is done by the vault.

At this point, the total deposit/debt in the lender strategy is 200/50. Suppose the price suddenly drops to $1, and the vault's position in the lender strategy gets liquidated before Bob's position is, selling 50 worth of borrow underlying for 100 worth of supply tokens (and ignoring the premium for the liquidator). Now the position of the protocol in the lender strategy is 100/0.

At this point, as the new `interestIndex` is 0: Bob does not have any debt towards the protocol anymore.

That does not seem fair at all. Pre liquidation, both Alice and Bob have assets worth $100 (Alice can withdraw $100 supply tokens, Bob can pay off his debt with the tokens he borrowed and withdraw $100 supply tokens). After liquidation, both Alice and Bob lose $25 worth of their deposits, but Bob in addition gets to keep the 50$ he borrowed.

There are different consequences of this observation:

1. The distribution is unfair and inconsistent with the behavior of the rest of the system. Note that the 25% that Alice loses in this example is not the cost of the liquidation itself (which is 0 in the

example), but rather a transfer of value from Alice to Bob. There is no reason for this transfer of value at all, and there is no other place in the system where such a similar thing happens

2. The example illustrates how Bob makes a (large) profit from the liquidation process. This provides incentives to Bob to actually cause the vault liquidation. For example, if his position is sufficiently large, Bob could try to force the liquidation of the vault by opening more debt positions and trying to manipulate the price in exchange for a considerable profit (in the form of transfer of value from other users)

Recommendation: Liquidation of the vault should, as much as possible, lead to the same distribution of losses as liquidation of individual users within the vault would have caused.

Severity: High
Resolution: The issue was later addressed in `f19e`. The new approach is discussed in the our audit of that commit.

IT2. burnBuffer does not diminish balanceOfAtIndex [high] [resolved]

Just as the `mint`, `burn` and `mintBuffer` functions, the `burnBuffer` function should update the `balanceOfAtIndex` value to reflect the new amount of available tokens. Failing to do so will skew all further calculations.
Recommendation: Add: `balanceAtIndex -= amount`; after line 122.
Severity: High
Resolution: The issue was resolved as recommended.

IT3. calcNewIndex may fail to return the new index, or revert [high] [resolved]

The `calcNewIndex` function has two if-clauses that seem to be meant to handle two mutually exclusive cases:

```
if (hasBeenLiquidated) {
      … handle the case where the new balance has decreased… }
if (balancePrev > 0 && balancePrev < balanceNew) {
      … handle the case where balance has grown … }
```

In these equations, the values for `balanceNew` and `hasBeenliquidated` are taken from the lender strategy, while `balancePrev` is based on the value of the balance when the last snapshot was made. It is important that the `hasBeenLiquidated` flag is true precisely in all cases where `balanceNew < balancePrev` for this to work. For if (A) `hasBeenLiquidated` is false but `balanceNew < balancePrev`, then the index will not be updated, and users can withdraw more underlying tokens than

are actually available in the strategy. If instead (B) `hasBeenLiquidated` is true, and `balanceNew > balancePrev`, the function will revert, and so will basically all user interactions with the vault.

Both these cases can actually occur, at least in theory. Consider the implementation of `borrowBalanceDetails` in `StrategyCompoundBase`, which returns the most recent balance and the value of the `hasBeenLiquidated` flag as:

```
uint256 balance = borrowBalance(borrowAsset);
return (balance, borrowPrincipal > 0 && borrowPrincipal > balance);
```

Here, the `borrowPrincipal` is the value of the debt as it is stored in the strategy, expressed in cTokens, and is updated on borrow and repay, while the returned `balance` is based on the `borrowPrincipal` times the accrued interest.
Suppose now that in the previous block someone has called snapshot - i.e. `balancePrev` reflects the value of `borrowBalance()` in the previous block. Scenario (A) could happen if a liquidation happened in Compound for a value of tokens that is less than accrued interest. Our case (B) would happen when the balance was already below the `borrowPrincipal` in the previous block, and in the current block has slightly grown because of accrued interest.
Recommendation: Replace the `hasBeenLiquidated` flag with a check that `balanceNew < balancePrev`. This will make the code easier to read, and reduce the possibility of errors in the implementations of strategies.
Severity: High
Resolution: The issue was resolved as recommended.

IT4. Consider removing the user interest accounting [low] [resolved]

For each account, the contract manages a mapping from addresses to numbers called `userInterest`, which more or less tracks the amount of difference from the principal a user has accrued in “not paid off debt”.
It seems to us that the only place in the contracts where this value is meaningfully read is in the `_repayVaultDebtInterest` function. This function is part of the `harvest` function, and regulates which part of the harvested earnings should go to paying off the vault’s debt (namely the amount of interest paid that was not paid off earlier), and which part should instead be deposited in the farming strategy.
In other parts of the system (and specifically in the rebalancing functionality) the ideal proportion of assets to be borrowed and farmed depends on the value of `targetThreshold` and the current relative value of deposit token versus borrow token. This is clear and straightforward. It is unclear to us why the part of the vault debt that is being tracked in the `userInterest` mapping gets precedence here and is paid off first also in situations in which rebalancing would result in a higher debt position.

Recommendation: We suggest removing the `userInterest` mapping and all related bookkeeping functions, as well as the vault interest repayment logic. This will save gas and reduce the complexity of the code considerably.
Severity: Low
Resolution: The issue was resolved as recommended.

IT5. getTotalInterest returns the wrong value [low] [resolved]

In line 278, the return value of the `getTotalInterest` function is calculated as follows:

```
totalUserInterest -= balanceStored - balanceNow;
```

This is wrong: as `totalUserInterest` at this point is always equal to 0, this will typically fail with an underflow error.
Recommendation: Replace the wrong line with:

```
totalUserInterest = userInterest[account] - (balanceStored - balanceNow);
```

Also, add a test for this case.
Severity: Low
Resolution: The issue was resolved. The `getTotalInterest` function was removed along with the entire user interest logic, as was recommended in IT4.

rToken.sol

RT1. transferFrom does not lower the allowance [high] [resolved]

The `transferFrom` function checks the owner's allowance and then transfers the tokens, but does not decrease the allowance, which means any allowance would give the account that received it access to the entire user balance.
Recommendation: Decrease the allowance of the caller on `transferFrom`.
Severity: High
Resolution: The issue was resolved as recommended.

LiquidatableManager.sol

LM1. Liquidators may lose money on liquidation [medium] [resolved]

When calling

```
supplyToken.transfer(
      usersForLiquidation[i],
      msg.sender,
      supplyLiquidatableAmount
);
```

Because of the way `transfer` is implemented (see also C11), this function is not guaranteed to actually transfer `supplyLiquidatableAmount` tokens - if the `usersForLiquidation[i]`address has a lower balance, then that balance will be transferred. This can happen if the value of the borrowed tokens to be liquidated, together with the liquidation bonus, is higher than the total amount of supply tokens deposited by this user. If that happens, the liquidator may operate at a loss - i.e. pay off the user's debt but not be compensated for it.
Although this situation is not very likely to occur, if it does occur it will be under extreme market circumstances. It is important precisely in those cases that liquidations happen effectively, and having liquidators incur a hard-to-manage additional risk is not recommended.
Recommendation: Implement a policy of "liquidate what you can" - if the user's balance of supply tokens is not enough to cover her debt, then at least the part of the debt that can be covered by her assets should be paid off. The debt that remains is bad debt, and there must be some policy in place to write off such debts. Currently, the remaining debt will be effectively transferred to all other holders of borrow tokens (the vault and users that have borrowed) in the system - but that may not be the most desirable behavior.
Severity: Medium
Resolution: The issue was resolved. Since the `transfer` function is now implemented as a standard ERC20 transfer, the transaction will revert if the user does not have enough balance for the liquidation amount.

LM2. Multiple liquidations fail if total liquidation amount is bigger than the buffer capacity [medium] [resolved]

In `liquidateUsers()`, the debt tokens of users are transferred in a loop to the buffer, from which they are later repaid to the lending strategy. If during the loop the amount of tokens in the buffer comes to exceed the `bufferSize`, the next iteration of the loop will revert when it calls `transferToBuffer`, as

that will call the `_userSnaphost` -> `snapshot` -> `calcNewIndex`, which will underflow when it will try to subtract the balance in the buffer from the buffer size (line 300):

```
uint256 bufferRefill = bufferSize - super.balanceOf(address(this));
```

This means that liquidations for a total amount bigger than the size of the buffer of the borrow token will revert if the size of the buffer is exceeded before the last liquidation in the loop.

In addition, due to the limits on liquidation (see LM4), there can be a case where a user's position is not big enough to be liquidated alone, but is bigger than the buffer size, and so will not be possible to liquidate until another position (or sum of positions) is liquidatable where that position is smaller than the buffer size, but at the same time big enough to be in total with the first position bigger than the minimum liquidation amount. This can potentially hinder closure of positions and cause the vault to be at a higher risk of liquidation in the lender protocol.

Recommendation: Burn the liquidated borrow tokens instead of transferring them to the buffer, then after the loop use the `totalRepayAmount` to calculate the repayment of the debt and refill of the buffer. Write a test for this scenario.
Severity: Medium
Resolution: The issue was resolved. In `calcNewIndex`, if the buffer balance exceeds the buffer size, the `bufferRefill` will be set to 0, and will no longer underflow.

LM3. Use of minUsersToLiquidate adds unnecessary risk [info] [not resolved]

Efficient and fast liquidation of under-water positions is important to avoid creating bad debt, which is a risk for all users and the system as a whole. The contract implements a restriction on liquidation: liquidation can only happen if more than `minUsersToLiquidate` users have an unhealthy position that can be liquidated, or the total value of the positions to be liquidated must be over `minRepayAmount`. These restrictions heighten the risk that bad debt is created. (The reason these restrictions are in place is to encourage arbitrageurs to liquidate positions in which the profit of the arbitrageur does not cover the gas costs of liquidation - but we do not think these restrictions help in this case).
Recommendation: Remove the condition.
Severity: Info
Resolution: The issue was not resolved. This was a conscious decision of Altitude, and they assured that at launch it will be set low enough as to not be a barrier for liquidators.

VaultConfiguration.sol

VC1. Vault owner can steal user borrow tokens on repayment [medium] [resolved]

The vault owner can, at any time, call the `setBufferConfig` and increase or decrease the buffer size. When increasing the buffer, borrow tokens will be taken from the `bufferCreditor` address the owner specifies and sent to the vault, and when decreasing the buffer, the tokens will be sent from the vault to the creditor address.
This means that when a user approves `borrowUnderlying` tokens to the vault with the intention to repay a debt, the owner of the vault can call the `setBufferConfig` and increase the borrow buffer size by the amount the user approved, and with the user address as the `bufferCreditor`. This will send the user's tokens to the vault. The owner can then call `setBufferConfig` again and decrease the borrow buffer by the same amount, this time setting themselves as the creditor. They will thus receive the user's tokens, effectively stealing them.
Recommendation: Require that the buffer creditor first calls a function to accept this role, or hardcode the vault owner to be the buffer creditor.
Severity: Medium
Resolution: The issue was resolved. Assuming the vault was created by the VaultRegistry, only the registry can call the `setBufferConfig`, and in the VaultRegistry's call to `setBufferConfig`, it will now always pass the `msg.sender` as the `bufferCreditor`.

VC2. Superfluous allowance check on buffer increase [low] [resolved]

When increasing the buffer size, the vault will transfer tokens from the buffer creditor to the buffer by calling `transferFrom`. This call already checks that the allowance is sufficient, and so makes the check of allowance before (lines 110 - 118) superfluous. This is true also for the initial buffer setting in the `VaultRegistry.sol _initialiseVault` function (lines 170 - 178).
Recommendation: Remove the check that the allowance is sufficient for the transfer, as the `transferFrom` function will already check that.
Severity: Low
Resolution: The issue was resolved as recommended.

VC3. Liquidation threshold setter could potentially be misused [low] [not resolved]

The `setBorrowLimits` function allows the owner of the vault to control its liquidation threshold. However, if this threshold is set too low, it could potentially trigger immediate liquidation of all the vault borrowers. This power of the owner could be potentially misused or abused.

Recommendation: There should be stronger limitations on setting this value. One option is to base the liquidation threshold on the actual liquidation threshold used by the active lending strategy, instead of having it settable by the owner. Another option is to require this threshold to be close to the threshold of the active lending strategy, or only allow changing it as part of the lending strategy change process.
Severity: Low
Resolution: The issue was not resolved.

VC4. Remove unused state variables [low] [resolved]

The state variables `lenderStrategies`, `farmStrategies` and `bufferCreditor` are written to storage but are then never read and so could be removed.
Recommendation: Remove the `lenderStrategies`, `farmStrategies` and `bufferCreditor` variables.
Severity: Low
Resolution: The issue was resolved. Of the 3 state variables we have suggested to remove, only the `bufferCreditor` was removed. However the `lenderStrategies`, `farmStrategies` are kept with the intention to use them in a future version of the `MigrationDecisionMaker.sol`, without having to update the `VaultConfiguration.sol` when that new version is ready.

VC5. Add sanity checks for the values of the various thresholds [info] [resolved]

The vaults contain three different "threshold" definitions:

- `liquidationThreshold` determines the maximum amount of debt a single user can incur before being liquidated.
- `supplyThreshold` determines the maximum amount of debt a user can borrow.
- `targetThreshold` determines the ideal fraction of tokens that should be borrowed and deposited in the farm strategy, and is used for rebalancing the portfolio.

Currently, the only validation on these arguments is that their values are not greater than `1e18`.
Recommendation: Consider adding some further checks, such that `supplyThreshold < liquidationThreshold`, and perhaps that `supplyThreshold <= targetThreshold`.
Severity: Info
Resolution: The issue was resolved. There are now checks that both the `supplyThreshold` and `targetThreshold` are less than or equal to the `liquidationThreshold`.

VaultCore.sol

C1. If the vault cannot repay its debts, remaining debts are distributed over other borrowers and to last withdrawers [high] [resolved]

In line 181, in `_repayVaultDebt`, all the `borrowTokens` held by the vault are burned:

```
borrowToken.burn(address(this), type(uint256).max);
```

However, there is no guarantee that all the debts of the vault are actually paid at this point: if the deposits in the farm are not enough to cover the entire debt attributed to the vault, then the debt that was taken on to deposit in the farming strategy may not be paid off entirely at this point.
With the current logic, the remaining debt will be distributed over the remaining holders of borrow tokens the next time a snapshot is taken.
This may lead to their positions becoming unhealthy If their supplied collateral is not enough to cover the debt; in that case these users will be liquidated.
In the end, it may be possible that bad debt remains in the system - i.e. there will be outstanding debt in the lender strategy, but no borrow tokens that represent that debt. If that happens, this remaining debt is not distributed equally over the users of the system, but instead, the last users to withdraw will find themselves unable to withdraw their supply tokens from the lending strategy. If this is a considerable sum, this information may even lead to a bank run.

Note that a similar case occurs on rebalancing. If the `targetThreshold` is 0, then rebalancing will withdraw all borrow tokens from the farming strategy to pay off the outstanding debt. If the farming tokens do not suffice to pay off the vault's debt, then the last users to withdraw their funding tokens will be unable to do so.
Recommendation: The situation where the vault's debt cannot be repaid entirely should be handled explicitly, and some policy choice should be made about who is responsible for that debt. It seems that the current situation - where other borrowers, or "last withdrawers", are responsible for the vault's debt - is arbitrary as well as not enough to resolve the situation. We recommend searching for another solution, for example to distribute the bad vault debt on the basis of a user's "active assets" (cf. issue HH1)
We also think that the system should include a mechanism that allows for detection and to intervene as soon as, or even before, the farming deposits do not suffice to cover the vault's debts.
Resolution: The issue was resolved. The `_repayVaultDebt` was replaced by the `disableFarmMode` in the `FarmModeManager.sol`, which only burns from the vault the debt that it was able to pay off, and distribute the losses as a negative harvest on the users.

C2. commitUser should be called before parameter validations [medium] [resolved]

In the functions `transfer`, `transferFrom`, `_borrow`, and `_withdraw`, the parameters passed to the function are validated by the `HealthFactorCalculator` before `commitUser` is called.
This means that these checks are done before taking a snapshot to update the `interestIndex`. Instead, a snapshot should be taken before validating the operations, as there is a possibility that a user performs an action (transferring, borrowing or withdrawing) that may leave her in an unhealthy position after the operation.
Recommendation: Call `commitUser` before validating the action parameters in the `transfer`, `transferFrom`, `_borrow` and `_withdraw` functions.
Severity: Medium
Resolution: The issue was resolved as recommended.

C3. Transfer and transferFrom should not try to move the harvest joining block of the sender [medium] [resolved]

In the `transfer` and `transferFrom` functions, the `_moveHarvestJoiningBlockWhenPositive` function is called for both the sender and the receiver. This means that both receiver and sender will not get any share of the harvest that was generated before the transfer took place - i.e. if A sends 1 token to B during the last block of the harvest, both A and B will get (almost) no share of the harvest itself. In this case, it would be more convenient for A to first withdraw 1 token and keep her claim of the harvest over the entire harvest period and then let B deposit the token himself. Moving the harvest block for the sender of the tokens is not necessary - the `joinBlock` only needs to be set for the receiver, who would otherwise be able to claim a disproportionate amount of the rewards on the basis of her new balance. Note also that `_moveHarvestJoiningBlockWhenPositive` is not called on withdrawal for the same reason, and so it should not be called here for the sender of the tokens to make the behavior consistent.
Recommendation: Implement our recommendation for HM2, which will avoid the problem. If that is not possible, remove line 91 in the `transfer` function and line 130 in the `transferFrom` function:

```
91: _moveHarvestJoiningBlockWhenPositive(msg.sender);
130: _moveHarvestJoiningBlockWhenPositive(from);
```

As there is no need that the `msg.sender` renounces *all* the earnings of the harvest when she sends part of her funds to another user.
Severity: Medium
Resolution: The issue was resolved in commit `f19e`.

C4. Avoid automatically claiming rewards when entering safety mode [medium] [resolved]

As part of the process of entering safety mode, rewards from the farming and lending strategies are collected (line 160 - 163), and swapped for the borrow (underlying) token.
Considering that safety mode will usually be triggered in a moment of price discrepancy, this might not be desirable, since in that case, rewards might be swapped at an unreasonable price. In such a case, it may be better to avoid claiming the rewards until prices return to a normal range.
What is worse, the various implementations of `recogniseRewardsInBase` may revert in certain cases - for example if there is a lot of slippage - in which case it is altogether impossible to enter safety mode. This is more likely to happen in case the market is very turbulent - i.e. in exactly the case where safety mode would be needed.
Recommendation: Avoid calling the lender strategy `recogniseRewardsInBase` (line 160) and remove it as part of the farming strategy's `withdrawAll` (you can call it separately in other places where `withdrawAll` is used) when entering safety mode. Instead, allow the owner to perform the claiming manually so they can use their discretion to decide if the rewards should be claimed during safety mode.
Severity: Medium - in some cases safety mode can not be entered.
Resolution: The issue was resolved. Safety mode was replaced with disable farm mode, and will still try to claim the rewards as before, but will no longer revert if an error occurs in the process.

C5. Not all price-sensitive functions are disabled in safety mode [medium] [resolved]

Safety mode is intended to pause some operations in the vault in case there is a confusion or attack regarding the prices of the underlying assets.
In `_enterSafetyMode`, two functions are disabled: `borrow` and `claimRewards`.
It makes sense to disable all operations that can be called by end users and that depend on reliable price information.
Recommendation: In addition to `borrow` and `claimRewards`, disable the following functions as well:

- `withdraw` (wrong price information can let users withdraw more than what is warranted by their outstanding debt)
- `transfer` (wrong price information can let users transfer more tokens than what is warranted by their outstanding debt)
- `liquidateUsers` (wrong price information can get users liquidated even if their actual position is healthy)
- `depositAndBorrow`
- `repayAndWithdraw`

Finally, consider *not* disabling `claimRewards` as it does not depend on price calculations at all.

Severity: Medium
Resolution: The issue was resolved. Safety mode has been removed.

C6. Move joining block on repay only if repayment was successful [low] [resolved]

In the `_repay` function, the `_moveHarvestJoiningBlockWhenPositive` function is called even if the `userBalance` was 0 and so no repayment has taken place. This unnecessarily reduces the user reward, even if no repayment has been made.
Recommendation: Move the call to `_moveHarvestJoiningBlockWhenPositive` (line 460) into the if clause (ending at line 458) that checks if the `userBalance` is greater than 0 before performing the repayment. Please also refer to issue HM2 for a different type of solution.
Severity: Low
Resolution: The issue was resolved as recommended.

C7. Wrong amount emitted in deposit event [low] [resolved]

In the `_deposit` function, the `Deposit` event is emitted (line 241) with the amount as the `msg.value`. However, this will only work for the `VaultETH` contract, and will always be 0 for the `VaultERC20`. The event should instead emit the `amount` function parameter, which will be correct for both.
Recommendation: Use the `amount` function parameter instead of the `msg.value` when emitting the `Deposit` event.
Severity: Low
Resolution: The issue was resolved as recommended.

C8. Transfer and transferFrom should check that the receiver has existing balance before calling commitUser [low] [resolved]

In the `transfer` and `transferFrom` functions, the `commitUser` function is called for both the sender and the receiver, but in case the receiver does not have any supply tokens yet, it will be a waste of gas to call the `commitUser` function for it. Instead there should be a check like the one that exists in `_deposit` to only call `commitUser` if the receiver already has tokens, or just set the `harvestIndex` to the last harvest if not.
Recommendation: Instead of always calling `commitUser` on the receiver in `transfer` and `transferFrom` (lines 87, 118), perform the same check that exists in `_deposit` (lines 216 - 223) to avoid unnecessarily calling `commitUser`.
Severity: Low
Resolution: The issue was resolved.

C9. Amount approved might be too high when repaying vault debt [low] [resolved]

In `_repayVaultDebt()`, the strategy is assigned an allowance of the total `lenderDebt` amount of tokens. In case the vault's balance is lower than that, the actual amount that should be approved is the vault's balance. This could be an issue with tokens that require the present allowance to be 0 to call the approve function. In that case, if more tokens were approved than actually used, the allowance will still be greater than 0, and future calls that try to set the allowance, such as `_withdraw` and `_repay` could fail.
Recommendation: Call the `approve` function only after the calculation of the amount to transfer is done, and only approve the amount that is to be transferred.
Severity: Low
Resolution: The issue was resolved as recommended. Note that the `_repayVaultDebt` was replaced by the `disableFarmMode` in the `FarmModeManager.sol`.

C10. Token approval might fail for certain tokens [low] [resolved]

As mentioned in C9, the amount approved in the `_repayVaultDebt()` might be higher than the actual amount used. In this case, or any other case where a certain amount is approved but is not fully used by the contract it's approved for, there will be an issue with tokens that require the present allowance to be 0 to call the `approve` function (for example USDT). In that case, whenever more tokens are approved than actually used, the allowance will still be greater than 0, and future calls that try to set the allowance, such as `_withdraw` and `_repay` could fail.
Recommendation: After each time tokens are approved and the call to the target contract is done, reset the approval of the token to 0. We recommend doing this through the entire codebase whenever an unknown token is used.
Severity: Low
Resolution: The issue was resolved. All approval calls now first reset the approval to 0 before approving the new amount.

C11. Transfer and transferFrom functions do not respect ERC20 standard [info] [resolved]

A call to `transfer(to, amount)` will transfer either `amount` tokens, or the complete balance of the sender, whichever is less. This does not follow the ERC20 standard which specifies that:

> "the function SHOULD throw if the message caller's account balance does not have enough tokens to spend" (ERC20 standard; https://eips.ethereum.org/EIPS/eip-20)

Recommendation: Follow the ERC20 standard where possible, so that calls to `transfer` and `transferFrom` give predictable results.
Severity: Info
Resolution: The issue was resolved as recommended.

C12. Redundant re-assignment of amount on transfer and transferFrom [info] [resolved]

In the `transfer` and `transferFrom` functions, the `_validateTransfer` call returns either the amount specified, or if the user balance is less than that amount, it will return the user balance. This result is then saved as `maxAmount` and passed to the token instead of the amount originally specified. This is redundant since there is identical logic in the `_accrualTransfer` function of the rToken, which will already make the token transfers behave in this way.
Recommendation: Avoid saving the result of the `_validateTransfer` call and instead just use the `amount` parameter instead of the `maxAmount`. Also see C11 regarding this irregular behavior of the token.
Severity: Info
Resolution: The issue was resolved. The entire unusual behavior was removed as recommended in C11.

VaultEth.sol and VaultERC20.sol

VV1. lock() modifier not applied consistently [low] [partially resolved]

The `lock` modifier defined in `VaultETH` and `VaultERC20` is a simple reentrancy guard: a first function call sets a flag so that any other calls of functions modified by the lock modifier will fail. The modifier is applied to a number of state-changing public functions such as `deposit`, `withdraw` and `borrow`.
The lock modifier provides some protection against reentrancy, but it is not applied consistently. For example, functions such as `claimRewards`, `transfer` or `liquidateUsers` do not have these locks set, and so the current implementation of the lock mechanism does not avoid re-entrancy when using these functions.
Recommendation: Use the lock more extensively and apply it to all public state-changing functions. Consider as well to use OpenZeppelin's `ReentrancyGuard` contract instead of defining your own lock function.
Severity: Low
Resolution: The issue was partially resolved. OpenZeppelin's `ReentrancyGuard` contract is now used. Most functions are now marked as `nonReentrant`, but transfers are not marked as such.

VV2. Remove duplicate code in VaultETH and VaultERC20 [low] [resolved]

`VaultETH` and `VaultERC20` have a number of functions with identical code: `deposit`, `withdraw`, `borrow`, `repay`, etc.
Recommendation: Remove duplicate code, for example by including the definitions of these functions in `VaultCoreV1`. This will also make it easier to implement our recommendation at VV1.
Severity: Low
Resolution: The issue was resolved as recommended.

VaultRegistry.sol

VR1. setAllFunctionsPause does not pause all functions [info] [resolved]

The function `setAllFunctionsPause` will disable (or enable, it can do both, and the function is not very well named) a fixed list of functions. The intention here seems to be to disable all functionality that transfers funds from users. The list however is incomplete, and does not include all public-facing functions. Other candidates for the list are:

- `claimRewards`
- `rebalance`

Recommendation: Add the missing functions to the list, or document the function and explain why others but not these functions will be disabled.
Severity: Info
Resolution: The issue was resolved. `setAllFunctionsPause` was replaceId by `setProtocolPause`, which now also pauses `claimRewards`. It does not pause `rebalance`, but this is intentional to allow rebalancing in case the vault did not enter the `disableFarMode`.

ChainlinkPrice.sol

CL1. Oracle owner can manipulate price results [info] [resolved]

The owner of the oracle can call the `ChainLinkPrice.setAssetMap` function at any time to change the address used for querying the price of an asset. This makes the oracle unnecessarily manipulable by the owner, which can make any query return the price of any other pair. While having an asset map for

special addresses is important for supporting assets like WBTC and WETH, there is no reason to have this changeable after the oracle is configured and deployed; it only adds risk to the system.
Recommendation: Make the asset map configurable only in the constructor, so it is not manipulable later by the owner.
Severity: Info
Resolution: The issue was resolved in `f19e`

UniswapV3Twap.sol

UT1. Make fee tier configurable per token pair [low] [resolved]

In the contract, `FEE_TIER` is a global variable, which is used to determine which of the pools of a given token pair to query for the TWAP. For getting a reliable TWAP price quote, the pool that is queried should be as large as possible, and this differs per token pair (i.e. the largest pools for USDC/ETH charge a fee 0.3% or 0.05%, while the largest USDC/USDT charges 0.01%.
This unnecessary limits the useability of the oracle, not only because it will not report the optimal price, but some tokens may not have a liquidity pool in a given fee tier at all.
Recommendation: Make the fee tier settable per token pair.
Severity: Low
Resolution: The issue was resolved as recommended.

UT2. Oracle does not support price discovery along path [info] [resolved]

The `UniswapV3SwapStrategy.sol` strategy also allows for swapping along paths. But the `UniswapV3Swap` Oracle does not implement querying prices along paths. This will make it hard (or even impossible if a certain swappable token pair does not have a native liquidity pool) to use the oracle alongside the Uniswap swap strategy.
Recommendation: Integrate the native Uniswap functionality for querying prices along a path:
https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/OracleLibrary.sol#L168
Severity: Info
Resolution: The issue was resolved. There is now a loop that queries the exchange rates through the whole swapping path.

StrategyGenericPool.sol

SG1. Deposit fails if contract already has Curve LP tokens [medium] [resolved]

In line 100, the contract calls the `convex.depositAll` function, which will try to deposit the whole balance of Curve LP tokens owned by the contract. Yet in line 99, the contract approves Convex to use only the amount of newly minted Curve LP tokens from the deposit. This means that if the contract has already had any amount of Curve LP tokens before the deposit, then Convex will try to deposit this entire balance, while it is approved only for depositing the amount newly minted, and so the transaction will revert.
This allows anyone to prevent deposits into the strategy by sending any amount of Curve LP tokens directly to the strategy contract.
Recommendation: Before deposit, approve the entire balance of Curve LP tokens held by the strategy to be deposited into Convex. This will also save some gas, because the value of `actualLPMinted` does not need to be calculated.
Severity: Medium
Resolution: The issue was resolved as recommended.

SG2. Wrong minimum amounts parameter passed on deposit [low] [resolved]

In line 93 the deposit function declares a `minAmounts` array, which the documentation says is the "Minimum amount of LP tokens to mint from the deposit". However, this array is then passed in the `add_liquidity` functions as the amounts to deposit, and the amount passed as minimum to mint is 0.
Recommendation: It is not clear if this behavior is a mistake in the documentation and function names, or a mistake in the implementation. In case this is a mistake in the documentation, and the behavior intended is correct, we recommend removing the `minAmounts` variable, and instead use the `toDeposit` parameter in the `_curveDeposit` function to create the amounts array there. It is also important to update the documentation to avoid confusion.
Severity: Low
Resolution: The issue was resolved as recommended. The `minAmounts` variable was removed and replaced with a `toDeposit` parameter in the `_curveDeposit` function.

SG3. Amounts awaiting re-deposit are not deposited with normal deposits [low] [resolved]

In line 136, if the amount withdrawn is greater than the amount requested, funds will be re-deposited only if they are greater than a certain limit given by `outstandingRewardsLimit`, otherwise they will just wait in the contract to be re-deposited. However, since the deposit function does not deposit amounts that are already in the contract, but only amounts added, these funds awaiting to be re-deposited will only be deposited when another withdrawal happens where the total amount to be re-deposited is bigger than the threshold.
Recommendation: On deposit, pass the entire asset balance held by the contract to the `_deposit` function, not just the new amount added.
Severity: Low
Resolution: The issue was resolved. The amount to redeposit is now saved in a state variable, and is redeposited along with the next deposit. Yet it would be cheaper and more efficient to query the contract balance as recommended, instead of maintaining a state variable.

SG4. Withdraw may revert if calculation of amount is too expensive [low] [resolved]

The `_calcExactLP` function is used by the `withdraw` function to calculate the amount that should be withdrawn to satisfy the withdrawal amount requested. This function will revert if the calculation requires too many iterations, which will make the `withdraw` function revert as well. Yet there seems to be no reason for that, since the `withdraw` function may withdraw an incorrect amount also in other cases, and has the logic in place to either re-deposit if the amount withdrawn was too high, or send less than requested if the amount withdrawn was too low. It would therefore be safer and more consistent to return the best result that could be calculated, instead of reverting.
Recommendation: On `_calcExactLP`, instead of reverting if reaching the end of the loop, return the last `crvAmount` calculated.
Severity: Low
Resolution: The issue was resolved. The calculation was re-written in a way that does not throw an error like before.

SG5. Avoid sending rewards from the contract to itself [low] [resolved]

The `_recogniseRewardsInBase` function sends the rewards claimed to either to the vault, if the function was called from the `recogniseRewardsInBase`, or to the contract itself in every other instance where this function is used. There is no reason to send tokens already held by the contract to itself, and so it would make sense to remove the to parameter from the function, remove the transfer of

tokens, and instead only transfer tokens to the `msg.sender` in the `recogniseRewardsInBase` function after calling the `_recogniseRewardsInBase`.
Recommendation: Remove the to parameter from the `_recogniseRewardsInBase` function, remove the transfer of tokens at the end, and instead only transfer tokens to the `msg.sender` in the `recogniseRewardsInBase` function after calling the `_recogniseRewardsInBase`.
Severity: Low
Resolution: The issue was resolved as recommended.

SG6. Save constants in immutable variables [low] [resolved]

Some data, such as the address of the `crvRewards` contract, or the address of the Curve token, are read from the Convex contract each time they are needed. As these values cannot be changed, the code can be simplified and some gas can be saved by reading these values in the constructor and storing them in state variables.
Recommendation: Store constant values such as the address of the CRV and the `crvRewards` contract and the value of `crvDecimals` in an state variable, instead of reading them every time from external contracts.
Severity: Low
Resolution: The issue was resolved as recommended.

SG7. Add an option to skip claiming extra rewards [low] [resolved]

The Convex contracts contain functionality to skip the claiming of extra rewards, in case one of these reward tokens is problematic (for example, it could be USDC and it could blacklist the Convex contracts, or it could be a token that has no corresponding uniswapv3 pool, and so the swap strategy may revert when trying to swap such tokens).
Recommendation: Implement similar functionality for skipping the claiming and swapping of extra rewards. Specifically, add a state variable `claimExtraRewards` that is controlled by the `contractManager`, and use Convex's `getReward(address _account, bool _claimExtras` on line 209 in the `_recogniseRewardsInBase` function.
Severity: Low
Resolution: The issue was resolved as recommended.

SG8. Possible inaccuracy in calculation of balance available to withdraw [low] [new]

The function `calc_withdraw_one_coin` calculates withdrawal in a single type of coin, yet in the documentation of the `_curveWithdrawAmount` function, it says it is calculated for the withdrawal of a

single unit of the coin, which is not correct. This leads to a misunderstanding in the `balance` function, where the withdrawal is queried for a single unit of the coin, then the total balance is manually calculated from that, while it would be more accurate to pass the `crvAmount` to the `_curveWithdrawAmount` function in line 80, and just return the result (plus the existing balance) as the balance, instead calculating from a single unit.
Recommendation: In line 80, pass the `crvAmount` to the `_curveWithdrawAmount` function, then return the result plus the current balance. In other words, replace:

```
uint256 singleUnit = _curveWithdrawAmount(crvDecimals);
return ((crvAmount * singleUnit) / crvDecimals) + outstandingDeposit;
```

with:

```
return (_curveWithdrawAmount(crvAmount)) + outstandingDeposit;
```

Also, fix the documentation of the `_curveWithdrawAmount` function. Note that we also recommend replacing the use of `outstandingDeposit` with just the actual contract balance.
Severity: Low
Resolution: The issue was resolved as recommended

SG9. Unused function _curveWithdrawAll [info] [resolved]

The `_curveWithdrawAll` function is defined in the contract but never used and could be removed
Recommendation: Remove the `_curveWithdrawAll` function.
Severity: Info
Resolution: The issue was resolved as recommended.

Aavev2FlashLoanStrategy.sol

AF1. Anyone can steal all funds of a vault which has this strategy registered [high] [resolved]

The `flashLoan` function of the contract allows anyone to call it with any parameters. This means that it is possible for an attacker to directly call this function, with the same parameters as if this function was called from the vault's `migrateLender` function, except that the value of `newStrategy` is the address of a malicious contract. Aave's `flashLoan` function will then call back the strategy's `executeOperation` function, which calls the `executeFlashLoanLenderMigration` function on the vault address to migrate the funds to the contract controlled by the attacker. So the attacker will receive

all the supply tokens of the vault to an address which they control, without having to pass the check in `migrateLender` that is supposed to ensure the migration can only be done to an address approved in the `MigrationDecisionMaker`.
Recommendation: Make the contract inherit from `VaultOperatable`, and make the `flashLoan` function callable only by the vault. Also, to ensure the owner of the strategy cannot steal the funds by changing the vault address, you should either make the vault address in `VaultOperatable` immutable, or remove the `targetAddress` from the `FlashLoan.Info` parameters passed to the function, and instead make it always be the `msg.sender`, so only the vault will be able to make the flash loan call back to itself.
Severity: High
Resolution: The issue was resolved. The `targetAddress` was removed rom the `FlashLoan.Info` parameters, and instead the `initiator` address is used, while it is enforced that the `initiator` must be the `msg.sender`. In addition, a state variable has been added that is set when the `flashLoan` function is called, then checked and immediately reset in the `executeOperation` function. This will help ensure that only the vault can trigger the migration of funds.

AF2. Pass 0 address as onBehalfOf when taking the flash loan [info] [resolved]

In line 40, the AAVE `flashLoan` function is being called, with the `onBehalfOf` parameter set to `address(this)`. But since the `flashLoan` function will ignore that parameter anyway (because the mode is always passed as 0), some gas could be saved by just passing the `address(0)` instead.
Recommendation: Change `address(this)` to `address(0)` on line 45.
Severity: Info
Resolution: The issue was resolved as recommended.

AF3. Unused variable ADDRESSES_PROVIDER [info] [resolved]

The contract defines and initializes an `ADDRESSES_PROVIDER` state variable, but never uses it and so it could just be removed.
Recommendation: Remove the `ADDRESSES_PROVIDER` state variable.
Severity: Info
Resolution: The issue was resolved as recommended.

StrategyAave.sol

SA1. Supply principal is manipulable by anyone [high] [resolved]

On deposit, the `supplyPrincipal` is updated based on the total supply balance of the asset deposited. But since the deposit is callable by anyone, with any supply token, it is possible for an attacker to deposit any other token supported by the AAVE pool, which would make the `supplyPrincipal` be set to a value based on a different token than the vault uses. This allows the attacker to manipulate the `supplyPrincipal` variable, and also the `supplyBalanceDetails hasBeenLiquidated` check that is based on it. By manipulating the `supplyBalanceDetails hasBeenLiquidated` value, the attacker can in turn manipulate the calculations in `InterestToken.calcNewIndex`, and could make it revert or make incorrect calculations.
Recommendation: Follow the recommendation in G1 and avoid passing the supply and borrow assets as parameters of every call to the strategies, and instead save the correct values in the constructor and use these saved values. It is also recommended that only the vault will be allowed to call the deposit, by marking the function as `onlyVault`, since there is no reason for anyone else to be able to call it.
Severity: High
Resolution: The issue was resolved as recommended. The `deposit` function no longer accepts the asset to deposit as a parameter, and is now only callable by the vault.

SA2. Lender info returns wrong supply balance and borrow power [high] [resolved]

Per the documentation, the `pool.getUserAccountData` returns as its first value “the total collateral in ETH of the user”. Cf.
https://docs.aave.com/developers/v/2.0/the-core-protocol/lendingpool#getuseraccountdata
This value is saved as the `currentSupplyBalance` and is also used to calculate the `totalBorrowPower` using the `convertToBase` function. However, the `convertToBase` expects to get the value in the underlying supply token, not in ETH as is returned here. This means that, for vaults where the supply asset is not ETH, the returned values of both `currentSupplyBalance` and `totalBorrowPower` will be wrong, the latter heavily affecting the rebalancing logic.
Recommendation: As for `currentSupplyBalance`, its value should be converted to the base asset. And as for the `totalBorrowPower` calculation, the price in base asset should be calculated against ETH, and with 1e18 decimals, instead of using the supply asset in the calculation.
Severity: High
Resolution: The issue was resolved as recommended.

SA3. Anyone can make rewards get stuck in the contract [medium] [resolved]

The `redeem` function accepts a `borrowAsset` parameter, which is a token address to which it will try to swap all the tokens received as rewards. The `redeem` function can be called by anyone.
An attacker could intentionally call `redeem` with an asset address that is approved to trade against the reward token on the swap strategy, but that is not the one the vault uses, and swap the rewards for that token. This means that the rewards will not be claimable when the vault calls the `recogniseRewardsInBase`.
The only way to recover those rewards then would be for the owner of the contract to call `setVault` to an address that can call `recogniseRewardsInBase` with the token the reward was redeemed for, then swap it for the correct reward and send it back to the contract, though this could cause serious disruptions in the operation of the vault.
Recommendation: Follow the recommendation in G1 and avoid passing the supply and borrow assets as parameters of every call to the strategies, and instead save the correct values in the constructor and use these saved values.
Severity: Medium
Resolution: The issue was resolved as recommended.

SA4. Remove unnecessary approve of borrow tokens [medium] [resolved]

In line 117, there is a call to the `borrowToken.approve` function, which approves the amount of tokens to be borrowed to the pool contract. There is no reason to do that, as borrow tokens are being sent from the pool to the strategy, not the other way around. This call is then not only unnecessary, but might even cause failures to borrow if the token borrowed does not allow re-setting of an existing approval amount, like USDT.
Recommendation: Remove the approve call in line 117.
Severity: Medium
Resolution: The issue was resolved as recommended.

SA5. Staking token rewards are not claimed on redeem [medium] [resolved]

The `redeem` function calls the `stakedRewardToken.redeem` to redeem the rewards claimed in the `claimRewards` function. Yet it doesn't claim the rewards of the `stakedRewardToken`, which are also available to be claimed by calling the `stakedRewardToken.claimRewards`.
Recommendation: Add a call to the `stakedRewardToken.claimRewards` to collect all rewards available. See also SA6 on swapping of the rewards.
Severity: Low

Resolution: The issue was resolved as recommended.

SA6. Wrong token swapped on redeem [low] [resolved]

The `redeem` function calls the `stakedRewardToken.redeem` to redeem the rewards claimed in the `claimRewards` function, then tries to swap all the `stakedRewardToken.REWARD_TOKEN()` available for the borrow asset. Yet the asset redeemed by calling `stakedRewardToken.redeem` is not the `stakedRewardToken.REWARD_TOKEN()`, but the `stakedRewardToken.STAKED_TOKEN()`. Thus no rewards will be swapped (unless the `REWARD_TOKEN` is the same as the `STAKED_TOKEN`).
Recommendation: Check if the `REWARD_TOKEN` and `STAKED_TOKEN` are the same, and if not, swap the `STAKED_TOKEN` to get the `stakedRewardToken.redeem` rewards, and the `REWARD_TOKEN` to get the `stakedRewardToken.claimRewards` rewards (See also SA5).
Severity: Low
Resolution: The issue was resolved as recommended.

SA7. Allow claiming rewards even when supply balance is 0 [info] [resolved]

The can claim function limits claiming of rewards for only when the supply asset balance is not 0. This may be too strict, considering that some rewards might still be left to claim even after all tokens have been withdrawn from the strategy, which will then only be possible to claim when more tokens are deposited.
Recommendation: Remove the requirement in line 233 that `supplyBalance` must be greater than 0.
Severity: Info
Resolution: The issue was resolved as recommended.

SA8. Should reset approval of borrow token after repay [info] [resolved]

In line 149, there is a call to the `borrowToken.approve` function, which approves the amount of tokens to be repaid to the pool contract. But since the `pool.repay` function may not use the entire amount approved (as the total debt might be smaller than the amount approved for it) then the approval left for the pool might not be 0 at the end of this call. This could cause a revert when trying to approve the token again for the pool, in case the borrow token does not allow re-setting of an existing approval amount, like USDT.
Recommendation: Reset the amount approved for the pool to 0 after the call to `pool.repay`.
Severity: Info
Resolution: The issue was resolved. All approval calls now first reset the approval to 0 before approving the new amount.

SA9. Remove unnecessary reset of approval [info] [resolved]

In line 301, there is a call to reset the approval of reward token back to 0 for the swap strategy, yet line 302 calls revert, which means the entire transaction will be reverted, including the call to reset the approval which becomes just a waste of gas.
Recommendation: Remove the reset of approval in line 301 and just revert the transaction. Also, consider resetting the approval even if the transaction does pass successfully, just in case the swap strategy does not use the full amount approved.
Severity: Info
Resolution: The issue was resolved. The function no longer reverts on error, but returns it.

SA10. Use claimRewardsToSelf to claim rewards [info] [not resolved]

In line 255, the rewards are claimed by the contract to its own address by calling the `incentivesController.claimRewards`, but since the contract claims the rewards to its own address, it is more appropriate to use the `incentivesController.claimRewardsToSelf` function which claims rewards to the caller, instead of asking for an extra address parameter to send the rewards to.
Recommendation: In line 255, change the call from `incentivesController.claimRewards` to `incentivesController.claimRewardsToSelf`, and remove passing the `address(this)` as parameter in line 258.
Severity: Info
Resolution: The issue was not resolved.

SA11. Remove unnecessary debt token delegation to self [info] [resolved]

In line 108, there is a call to the debt token's `approveDelegation` function which delegates the tokens of the contract to itself. It is unclear why it is done, as there is no need to delegate tokens to their owner.
Recommendation: Remove the delegation of debt tokens owned by the contract to itself.
Severity: Info
Resolution: The issue was resolved as recommended.

SA12. Mark functions as external where possible [info] [partially resolved]

The `deposit`, `withdraw`, `withdrawAll`, `borrow`, and `repay` functions are declared `public` but are not used from within the contract, and so could be marked as `external`.

Recommendation: Mark `public` functions that are not used within the contract as `external`.
Severity: Info
Resolution: The issue was partially resolved. All functions recommended to be marked as external were marked as such, except the `deposit` function, which remains `public`.

StrategyCompoundBase.sol

SC1. enterMarkets call is not properly checked for success [low] [resolved]

The constructor calls the `comptroller.enterMarkets` function for both the supply and borrow tokens, but then only checks the success of the call for the supply asset. So if the `enterMarkets` call fails for the borrow token it will go undetected. This could cause issues later on.
Recommendation: Make sure that both the first and second values in the array returned from the `comptroller.enterMarkets` call are equal to 0.
Severity: Low
Resolution: The issue was resolved as recommended.

SC2. getInBase returns price of cSupplyToken's underlying asset, not of provided token [low] [resolved]

The function `getInBase(supplyAsset)` checks the price of the underlying asset of `cSupplyToken`, and multiplies that by the number of decimals of the `supplyAsset` parameter. This means that this function will return unexpected results when called with a `supplyAsset` that is not the underlying token of the `cSupplyToken`.
This problem is inherited by the `convertToBase` function. See also G1.
Recommendation: Remove, or in any case, ignore the value of, `supplyAsset` parameter from the `getInBase` function. Adapt `convertToBase` in a similar way.
Severity: Low
Resolution: The issue was resolved. The `getInBase` function will now revert when called with the wrong asset.

SC3. Wrong supply balance returned when total supply is 0 [low] [resolved]

The `supplyBalance` function will calculate the supply balance similarly to how it would be calculated on the contract itself, but keeping the function as `view`. When doing this calculation, if the

`totalSupply` of the token is 0, the `supplyBalance` returns `initialExchangeRateMantissa` from the Compound contract. It should return 0.
Recommendation: Return 0 instead of `initialExchangeRateMantissa` in line 170.
Severity: Low
Resolution: The issue was resolved. When the supply is zero, the `initialExchangeRateMantissa` is now used as the exchange rate instead of being returned as the final result.

SC4. Remove _getCashInternal [info] [resolved]

In both implementations inheriting from `StrategyCompoundBase`, the function `_getCashInternal` is implemented identically. It could just be replaced by a call to `CToken(cSupplyAsset).getCash()`.
Recommendation: Remove the `_getCashInternal` function and replace it with `CToken(cSupplyAsset).getCash()`.
Severity: Info
Resolution: The issue was resolved as recommended.

UniswapV3Strategy.sol

US1. Fees might be miscalculated in multihop swaps due to slippage [low] [resolved]

In the function `setSwapPair`, a variable named `totalFee` is calculated and stored with the token pair. However, if the path of the swap goes through different token pairs, it is impossible to calculate in advance what percentage of the original value will be paid in fees, as it is unknown in advance what price will be paid for the tokens at each hop (typically, because of slippage, the actual amount of fees paid will be lower than the number calculated, but also other price discrepancies could lead to a higher amount of fees).
The saved value for `totalFee` of a token swap pair is used in `getMinimumAmountOut` to calculate the minimum amount to get out of a swap, which combines this number of totalFees with the maximum amount of acceptable slippage in a trade. Concretely, this means that for swaps which are done with multiple hops, the fee calculation might be wrong and actual slippage paid in the trade (to arbitrators) may be higher than `maximumSlippage`.
Recommendation: Remove the distinction between “slippage” and “fees”: from the viewpoint of the trader, the distinction is irrelevant. Instead, apply our recommendation in US2 and make slippage configurable per pair.
Severity: Low
Resolution: The issue was resolved as recommended.

US2. Make Maximum slippage configurable per pair [info] [resolved]

Currently, the maximum slippage allowed for a trade is configured as a global variable. But since different pairs may have different market liquidity, it could be useful to have the option to configure the maximum slippage per pair instead of globally.
Recommendation: Have the maximum slippage configurable for each trading pair instead of a single global variable.
Severity: Info
Resolution: The issue was resolved as recommended.

US3. Remove SLIPPAGE_BASE [info] [resolved]

The `SLIPPAGE_BASE` can be set in the constructor to have any potential base for the slippage calculations, but this just adds unnecessary complexity, especially when setting the maximum slippage. It could instead be replaced with the `FEE_100` constant to avoid possible confusion in calculations.
Recommendation: Remove the `SLIPPAGE_BASE` and use the `FEE_100` (or a similar constant) instead.
Severity: Info
Resolution: The issue was resolved. `SLIPPAGE_BASE` is now a simple constant.

US4. Unused variable factory [info] [resolved]

The contract defines and initializes a `factory` state variable, but never uses it and so it could just be removed.
Recommendation: Remove the `factory` state variable.
Severity: Info
Resolution: The issue was resolved as recommended.

Appendix: Test Coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
common/	100	100	100	100	
ProxyExtension.sol	100	100	100	100	
RolesManageable.sol	100	100	100	100	
decision-makers/migration-decision/	100	100	100	100	
MigrationDecisionMaker.sol	100	100	100	100	
decision-makers/rebalance-decision/	100	100	100	100	
RebalanceDecisionMaker.sol	100	100	100	100	
decision-makers/safety-mode-decision/	95.92	91.67	100	96.36	
SafetyModeDecisionMaker.sol	95.92	91.67	100	96.36	197,199
libraries/	100	85.29	100	96	
AbsMath.sol	100	100	100	100	
DataTypes.sol	100	100	100	100	
Errors.sol	100	100	100	100	
FlashLoan.sol	100	100	100	100	
HarvestHelper.sol	100	87.5	100	100	
HarvestTypes.sol	100	100	100	100	
HealthFactorCalculator.sol	100	100	100	100	
RMath.sol	100	60	100	80.95	34,44,54,66
SwapErrors.sol	100	100	100	100	
libraries/uniswap-v3/	90.91	80	75	84	
FullMath.sol	100	100	50	50	120
OracleLibrary.sol	88.24	83.33	100	88.24	75,80
PoolAddress.sol	100	75	100	100	
TickMath.sol	100	100	50	50	98
misc/incentives/	66.67	50	80	71.43	
IncentivesManager.sol	66.67	50	80	71.43	55,57
oracles/	90.2	87.5	90	91.38	
ChainlinkPrice.sol	85.71	81.25	83.33	86.84	... 108,151,153
UniswapV3Twap.sol	100	100	100	100	
strategies/farming/convex/	97.89	100	90.91	96.94	
StrategyGenericPool.sol	100	100	100	98.82	165
StrategyMeta3Pool.sol	85.71	100	80	85.71	92
StrategyMetaPool.sol	83.33	100	80	83.33	83
strategies/flashloan/	100	50	100	89.47	
Aavev2FlashLoanStrategy.sol	100	50	100	89.47	63,69
strategies/lending/aave/	100	83.33	100	100	
StrategyAave.sol	100	83.33	100	100	
strategies/lending/compound/	100	81.58	100	99.19	
StrategyCompound.sol	100	70	100	95.24	88
StrategyCompoundBase.sol	100	85	100	100	
StrategyETHCompound.sol	100	87.5	100	100	
strategies/swap/	100	100	100	100	
SwapStrategyConfiguration.sol	100	100	100	100	
UniswapV3Strategy.sol	100	100	100	100	
tokens/	95.9	86.96	100	96.21	
InterestToken.sol	94.57	83.33	100	94.95	... 182,211,278
TokensFactory.sol	100	100	100	100	
rToken.sol	100	100	100	100	
rdToken.sol	100	100	100	100	
vaults/v1/	98.7	87.72	100	96.38	
VaultConfiguration.sol	100	96.88	100	98.94	97
VaultCore.sol	97.52	81.94	100	93.37	... 606,631,666
VaultFactory.sol	100	100	100	100	
VaultRegistry.sol	100	100	100	100	
VaultStorage.sol	100	100	100	100	
vaults/v1/ERC20/	100	100	100	100	
VaultERC20.sol	100	100	100	100	
vaults/v1/ETH/	100	100	100	100	
VaultETH.sol	100	100	100	100	
vaults/v1/extensions/groomable/	100	100	100	100	
GroomableManager.sol	100	100	100	100	
GroomableVault.sol	100	100	100	100	
vaults/v1/extensions/harvestable/	97.78	72.73	95.45	95.77	
HarvestableManager.sol	98.88	80.77	100	98.92	143
HarvestableVault.sol	95.65	61.11	93.75	89.8	... ,94,111,146

vaults/v1/extensions/liquidatable/	89.29	92.86	100	90.91	
LiquidatableManager.sol	85.71	90	100	86.96	111,113,118
LiquidatableVault.sol	100	100	100	100	
All files	97.85	86.99	97.82	96.51	