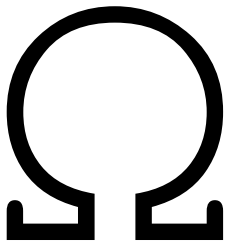
Altitude v1.1

Audit Report

October 9, 2023 (preliminary version)
January 22 (update with fixes)
March 21, 2024 (final update)



Team Omega

teamomega.eth.limo

Summary

Altitude has asked Team Omega to audit their smart contract system.

On October 23, 2023, we wrote a preliminary version of the current report.

In January 2024, we did a review of an updated version of the code, as a part of which we also reviewed the resolution of the issues from this report.

We have audited these contracts before, in the last half of 2022. The code has changed significantly since our audit of last year: we checked the fixes that were made of the issues noted in the previous report, and updated that report (see details below in the “Scope” section).

We found a number of new issues, which are described in the current report. Specifically, we found **2 high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **8** issues as “medium” - these are issues we believe you should definitely address, even if they do not lead to loss of funds. In addition, **6** issues were classified as “low”, and **1** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Severity	**Number of issues**	**Number of resolved issues**
High	2	
Medium	8	
Low	6	
Info	1	

Scope of the Audit

We audited the code from the following repository:

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

We audited the code in this repository in the second half of 2022.

The present audit reports regards the changes that were made after our audit report Altitude Audit Report 2022 - Final Report was submitted. Specifically, this audit report regards the changes that were made between commit `25dd16ccd13164f9c2b7ba63743bd07f049813d2` (the final commit of the previous report from October 22, 2022) and commit `f19e2e5edeb87e5e37345932cede3a0e99709f23`

These changes regard:

- Error handling: Moving to revert for all errors and to improve maintainability no longer using an Errors.sol file
- Farm Slippage Check: Introduction of farmStrategy slippage checking to avoid a Yearn type exploit https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md
- Vault Liquidation handling: Addressing of Vault Liquidation edge-case where individual account may benefit from the vault being liquidated
- Introduction of Rate-Limits: Extracting access-control logic to a separate contract and introducing a rate-limit to make it harder to quickly extract large sums from the protocol through an exploit
- Farm Drop monitoring: Allow for monitoring of farmStrategy value and entering farmMode on sudden reductions in value (e.g. de-peg or external protocol issues)
- Contract Size & Simplicity: Optimisations for contract size and/or simplicity

There is new functionality that is out of scope of this audit:

- User Migration Tool: Tool to allow users to migrate from an existing lending pool to Altitude
- Lending Strategies: New lending strategies for AaveV3 and CompoundV3 respectively

Specifically, the following files in the repository were **not** part of this audit:

- All the files in the `migration` directory
- The files in `contracts/lending/strategies/lending/Aave/v3/`
- The files in `contracts/lending/strategies/lending/compound/v3/`

Resolution

We reviewed 0c8f211e0ae587ba3635ace9fa87e4adae2d80ef in the report at https://docs.google.com/document/d/1TuaCggKCA4UyTSN_KecdR0Iwcwn6j7vlYDP1olu8iZo/

Those issues that were resolved are marked as such below each issue.

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

Our general impression of the code base is that it is overly complex in certain cases. This makes the code difficult to read in some places, and makes it harder to audit the code and reason about it properly.

We are offering here a number of suggestions for improving the code base.

- The code uses a way of handling proxies in which each function is explicitly mapped to the proxy. This pattern is very verbose (in practice, all definitions of function signatures are defined three times - in the interface, in the proxy contract, and in the implementation contract. Moreover, the pattern does not allow for ways to separate the storage of the different implementation contracts, so they all copy the entire storage space, and will make it harder to upgrade the various parts of the system in a modular way. There are more or less established proxy patterns, for example those of the OpenZeppelin library, which would make your code easier to read and less error prone.

- Function definitions are often spread over different files in a way that often seems arbitrary. Between interfaces and implementations, the function signature for `disableFarmMode` appears *six* times. The core function definition is spread over three different files:
    1. `FarmModeManager.disableFarm` mode checks the health, and in certain cases will then call
    2. `VaultCore.disableFarmMode`, which creates a snapshot, and then calls:
    3. `FarmModeVault.disableFarmMode`, which forwards the call to
    4. `FarmModeManager.disableFarmMode` where the rest of the logic happens

  These patterns make the code hard to read, and also lead to errors such as VC1.

- Try to avoid mixing the usage of signed and unsigned integers. In some cases “costs” and “earnings” are saved as unsigned integers, in other places, losses are defined as negative numbers, and there is a lot of casting back and forth.

- Give the SupplyToken a clear semantics. The supply token is currently used for two different purposes: as the balance of a user, it represents the amount of supply underlying tokens the user can withdraw from the system (if their debt is paid off), and as the balance of the token itself, which represents the amount of underlying tokens that are present in the buffer. This double meaning has repercussions in many parts of the code

- Give the debt token a clear semantics. The current use of the debt token has the same double usage as the supply token in that it is used to represent the user’s debt, and to mirror the buffer usage (although in this case it represents the amount of underlying debt tokens that are _not_ present in the buffer). In addition, as the balance of the vault, it seems to represent the amount of underlying tokens deposited in the farming strategy.

- There is a lack of separation of concerns and abstraction in the code, which makes the code hard to read and to follow. We give two examples where we believe this is very clear:

- The logic for handling the buffer is replicated in some (but not all, cf HM3) of the functions that withdraw or deposit funds, and appears in different forms in many places in the code when calculating global amount of assets and debt. Such logic should typically be kept in a single place, and handled in a way that is transparent to other functions.
- Another example where a better separation of concerns could considerably simplify the code is in the `commitUser` logic. This function does two very different things. It calculates the current user's positions and commits the result of the calculation on-chain (mostly by updating the user's balance). But at the same time it tries to settle eventual debts of the user by withdrawing funds from the farming strategy, it may mint new tokens to adjust the buffer, and in general change the global state of the system.

- The logic for Access Control uses an approach where access needs to be set up and managed per function signature, instead of using one of the available libraries for access control like OpenZeppelin's role based system. This approach is non-standard, and complicates both the code and the setup and management of the system on and after deployment. Also, the decision making code for changing strategies uses a kind of a customly written multisig, while it could just as well be integrated with the normal role based permission and use an external multisig as the manager of that role, maintaining better consistency and readability while reducing the potential attack and error surface.

- A recurring pattern which makes the code hard to read in some places is to reuse variable names for different purposes in the same function, which may save a negligible amount of gas, but makes the code very hard to read and reason about as it repurposes variable into usage that cannot be assumed from their naming or previous values.

AccessControl.sol

AC1. userMinDepositLimit may not be properly enforced [low] [resolved]

The check enforcing that `userMinDepositLimit` is respected checks that the user's balance is not 0, or that the amount deposited is at least the `userMinDepositLimit`.
A user can circumvent this check by receiving some supply tokens by a normal transfer, which will make the user balance not 0, and so the check will pass. Now the user can deposit any amount they want, which, combined with the current balance, will still be lower than `userMinDepositLimit`.
Recommendation: Replace the check in line 208 to check that current balance + amount deposited is greater than `userMinDepositLimit`.
Severity: Low
Resolution: The issue was resolved as recommended

AC2. Debt of sanctioned users will remain stuck in the system [medium] [resolved]

When a user is added to the sanctioned list, they cannot withdraw funds, their debt cannot be repaid, they cannot be liquidated, and they cannot be committed for harvest or vault liquidation purposes. Essentially, their funds and position remain "stuck" in the system unless they ever get un-sanctioned. This is a problem, because it means that if such a user's position becomes unhealthy, it may leave bad debt in the system.
Recommendation: Depending on your regulatory situation, either allow sanctioned users to only repay and withdraw. You might also want to allow liquidating such users, which will require allowing to commit them. Another approach is to implement functionality to remove a user from the system (essentially by liquidating their entire borrow position and sending them the supply funds)
Severity: Medium
Resolution: The issue was resolved as recommended: the debt of sanctioned users can now be repaid by themselves, or by any other non-sanctioned user. Also, sanctioned users can be liquidated.

AC3. validateInjectSupply should use onlyRole modifier [info] [resolved]

The `validateInjectSupply` function just duplicated the code of the `onlyRole` modifier with no reason and could use the modifier itself instead.
Recommendation: Remove the code duplication and use the `onlyRole` modifier.
Severity: Low
Resolution: The role system was refactored and this issue is not relevant anymore.

CommitMath.sol

CM1. userHarvestUncommitedInterest could be removed [low] [resolved]

The `userHarvestUncommitedInterest` holds a value that is always and practically immediately added to the `up.borrowBalance` and deleted from the `userHarvestUncommitedInterest`. There is no reason to keep the `userHarvestUncommitedInterest` then, and you could remove the it from the code by changing line 226 to:

```
cp.up.borrowBalance += userHarvestChange.value;
```

Also since `userHarvestUncommitedInterest` always resets to 0, `user.uncommitedInterest` in the HarvestableManager will also always be 0, and can be removed from the code as well.

Recommendation: Remove `userHarvestUncommitedInterest` and `user.uncommitedInterest` from the code and replace line 226 as suggested above.
Severity: Low
Resolution: The function was removed, as recommended

ChainlinkPrice.sol

CP1. wstETH is calculated from ETH price, not stETH [low] [resolved]

The oracle will try to calculate the `wstETH` price based on the price of `ETH` - it assumes there is a direct relation between the price of ETH and the price of stETH. However, deviations between the ETH and stETH price of up to 10% have already occurred in the past, and are likely to happen in the future as well, which will cause the oracle to return a wrong price.
Recommendation: It is better to get the price of stETH directly instead of relying on the price of ETH. You can use the functions such as `getStETHByWstETH` from the wstETH contract to derive the price of wstETH from the stETH price.
Severity: Low
Resolution: The price of `wstETH` is now derived by querying the Chainlink Oracle for the price of `stETH`, and multiplying that by the amount of pooled `ETH` each `wstETH` token represents. This should result in a good approximation of the market value of `wstETH`

DebtToken.sol

D1. userUncommittedEarnings is not rebased in _balanceOf() [medium] [resolved]

In the `_balanceOf()` function, the balance of the user given a certain commit position is calculated as the borrow balance, minus the uncommitted harvest costs, plus the uncommitted earnings. In the calculation, the `borrowBalance` and the `userHarvestUncommittedCosts` are rebased relative to the most recent index, but `userHarvestUncommittedEarnings` is not. This means that that balance will not take into account any changes in the value of the shares of the earnings, and so may over- or under-report the actually available user balance.
This is important, because `_balanceOf()` is used to determine how much a user can withdraw, or to calculate the health of a user's position.
Note that the same holds in `HarvestManager._applyCommitData`.

Recommendation: Also rebase the earnings with respect to the index, both in `_balanceOf()` as in `_applyCommitData()`.
Severity: Medium
Resolution: The team comments that “We have removed the need for `uncommitted costs` so for this the issue is no longer relevant.

Regarding userHarvestUncommittedEarnings, these are intentionally not rebased as they don't accumulate interest. Currently we capture the amount at time of harvest and distribute this amount to users, but interest doesn't accumulate on this amount.”

HarvestableManager.sol

HM2. Inappropriate use of delegateCall [low] [resolved]

On lines 445ff, there is a `delegateCall` to the current contract. This is uselessly verbose, and reverse with a misleading error message:

```
(bool success, bytes memory data) = address(this).delegatecall(
    abi.encodeWithSelector(IHarvestableVaultV1.reserveAmount.selector)
);
if (!success) {
    revert HM_V1_FAILED_EXTERNAL_CALL();
}

uint256 maxAmount = abi.decode(data, (uint256));
```

Recommendation: Replace these lines with `uint256 maxAmount = reserveAmount()`, and move the code for `reserveAmount` to the from `HarvestableVault` to the `HarvestableVaultManager` contract, which is the place where the logic is defined.
Severity: Low
Resolution: The issue was resolved.

HM3. claimRewards may revert and will not pay off existing debt [low] [not resolved]

The function `claimRewards()` takes the user’s claimable earnings and uses that to pay off the debt of the user. This is an internal adjustment of balances, and no tokens are actually transferred.

If there are additional rewards that remain, these will be withdrawn from the farming strategy and sent to this user. If this fails (for example because the farming strategy does not have enough tokens to cover the debt) the call to `claimRewards` will revert.

This is not desired behavior. The user's debt should be paid off with the rewards, even if the farming strategy does not have enough tokens available to send the remaining rewards

Recommendation: Do not revert in case the farming strategy cannot cover the rewards. Also, use the buffer here as you do in other places where tokens are moved from and into the farming strategy.
Severity: Low
Resolution: The issue was not resolved. Altitude has acknowledged the issue and decided not to resolve the issue.

HM4. Existing rewards may be blocked from being harvested [medium] [not resolved]

The `harvest` function checks if `vaultActiveAssets` are positive, and will revert if not. This means that if there are lender rewards, or if there are farming rewards generated before the `vaultActiveAssets` became negative (and was rebalanced), then these rewards will not be possible to harvest. This means that users can prevent the vault from being harvested by taking out debt, negatively affecting the rest of the users. Users who want to withdraw their funds will have to forfeit their pending rewards for the upcoming harvest.
Recommendation: Have a clear separation of concerns. We conjecture that the condition that vaultActiveAssets must be non-zero was put in place because, later, user rewards are distributed on the basis of their share in vaultActiveAssets. But such considerations of how to distribute the gains from the harvest should not be part of the harvest logic itself.
Severity: Medium
Resolution: The issue was not resolved - Altitude has acknowledged the issue.

HarvestTypes.sol

HTx. In UserHarvestData some fields are not used [low] [resolved]

The field for `harvestEarnings` and `harvestCosts` are only written, but never read, and can be removed to improve complexity and gas costs
Resolution: The issue was resolved as recommended.

LiquidationManager.sol

LM1. realUncommittedEarnings are not respected in _withdrawVaultBalance [medium] [resolved]

In `_withdrawVaultBalance`, the following lines define logic to limit the withdrawal to an amount that leaves an amount of `realUncommittedEarnings` in the farm strategy:

```
if (maxToWithdraw >= harvestStorage.realUncommittedEarnings) {
   maxToWithdraw -= harvestStorage.realUncommittedEarnings;
}
```

However, this omits the case in which `maxToWithdraw` is smaller than `realUncommittedEarnings`: in that case, the entire balance is withdrawn and no tokens will be left to cover the `realUncommittedEarnings`, and users may lose funds
Recommendation: Add the else clause and set `maxToWithdraw = 0`
Severity: Medium
Resolution: The function was moved to SupplyLossManager.sol. The issue was resolved as recommended

LM2. Vault's balance is reset with the wrong value in _withdrawVaultBalance [medium] [resolved]

The final lines in `_withdrawVaultBalance` look like this:

```
borrowToken.setBalance(
    address(this),
    vaultBalance - withdrawn,
    borrowIndex,
}
```

Here, `withdrawn` represents the amount of `borrowUnderlying` tokens that were withdrawn from the farm strategy. The variable `vaultBalance` is a parameter to the function (and is commented as "the amount to be withdrawn"), and is not necessarily the current balance of borrowTokens of the vault. This function is only called in one place, namely in `snapshotVaultLiquidation`, where the value passed to the `vaultBalance` is the balance from immediately before the liquidation happened (although in some cases the values stored there are from immediately after the liquidation, but that is not relevant to this issue). The point is that in the case of liquidation, the value of `vaultBalance` will typically be higher than the actual amount of `borrowTokens`, and the function will withdraw tokens from the farming strategy and leave an amount of `realUncommittedEarnings` in the farming strategy.

This value has no relationship with the value of `vaultBalance - withdrawn`, which is almost surely not the same number as the amount of tokens left in the farming strategy.
Recommendation: It seems that the purpose of this function is to withdraw all tokens from the farming strategy (except those reserved for `realUncommittedEarnings`). We recommend you implement that.
Resolution: The vault's balance of the borrow token is now set to 0.

StrategyCompoundBase.sol, StrategyAaveV2.sol

SCB1. hasBeenLiquidated can be manipulated [high] [resolved]

The `hasBeenLiquidated` function is meant to check whether the vault's position in the lender strategy has been liquidated. This is then used as a flag for triggering an "emergency mode" of the system. However, the function can be easily manipulated by an attacker, which could wrongly trigger such liquidation emergency mode at will. Since the function checks that either the borrow balance, or the supply balance, were reduced in the lender since the last action of the strategy, an attacker could use the `repayBorrowBehalf` functionality of Compound to repay a small amount of the protocol's debt, which will then cause the borrow balance to be lower than what was saved in the strategy, and so trigger a false liquidation.
Also Aave allows for repaying on behalf of others (https://docs.aave.com/developers/core-contracts/pool#repay). Aave also allows for supplying tokens on behalf of others (https://docs.aave.com/developers/core-contracts/pool#supply)

Recommendation: Given the modular nature of the system, the inclusion of a function such as `hasBeenLiquidated` in the interface is fundamentally problematic. There is no guarantee that future integrations of lending providers will provide the functionality to assess if a position was liquidated or not - as indeed is the case with Compound and Aave.
We recommend rethinking your logic here, and in particular not "go into emergency mode on liquidation". Liquidation is an unfortunate, but controlled and predictable event, and is designed to result in a lending position that is healthy.
That said, an easy but partial fix for the Compound and Aave case is to check only if the supply balance has decreased. That will remove the "false positives". It does not seem to be possible to remove the "false negatives" (where the function returns true even if the vault has been liquidated).

Severity: High
Resolution: The check now checks if the amount of supply tokens has unexpectedly diminished, which is what we recommended.

StrategyCompoundV3.sol

SCV1. Liquidation bonus is not calculated correctly [medium] [resolved]

On line 275, the liquidation bonus is calculated as follows:

```
liqBonus = data.scale + (data.scale - data.liquidationFactor);
```

This is not correct, as the `liquidationFactor` is "an integer that represents the decimal value scaled up by 10 ^ 18", while `scale` is "An integer that equals 10 ^ x where x is the amount of decimal places in the collateral asset's smart contract." So, if the collateral is, for example, USDC, which has 6 decimal places, this logic will underflow.
Recommendation: Read the documentation in
https://docs.compound.finance/helper-functions/#get-asset-info
Severity: Medium
Resolution: The decimal logic now is sound.

StrategyGenericPool.sol

SGP1. getMinimumAmountOut will overestimate the result [medium] [resolved]

The function `getMinimumAmountOut` tries to give an estimate of the minimum amount of lpTokens one would get when depositing tokens of the borrowAsset, or, vice versa, the amount of borrowAssets one would get given a number of lpTokens.
The function uses curves `get_virtual_price` to determine the price of the lpToken, which reports the price of the lpToken in units of the cheapest token in the pool. The function then returns the amount of lpTokens one would get when depositing the cheapest token (or, vice versa, the amount of cheapest token one would get when redeeming lpTokens).

In other words, the price of the `borrowToken` is not taken into account - instead, a price based on the value of the cheapest token in the pool is returned. As the function is used for estimating the price of borrowTokens, this is a problem: the amount of lpTokens one gets in exchange for borrowTokens is higher than the actual one (which will make it more likely that functions that use this as the price estimate will revert); while the estimated conversion from lpTokens into borrowTokens is underreported (which will make it more likely that your trade will be front-run)
Recommendation: Take the price of the borrowAsset into account when calculating `getMinimumAmountOut`.
Severity: Medium
Resolution: The issue was resolved by using a number settable by admins

UniswapV3Strategy.sol

UVS1. Wrong swap pair data used in swapOutBase and getMaximumAmountIn [medium] [resolved]

The `swapOutBase` and `getMaximumAmountIn` functions both get the `SwapData` of the pair they are asked about, and will revert if the pair was not already configured with the `setSwapPair` method. However, both `swapOutBase` and `getMaximumAmountIn` check for the wrong pair, as they check for the `swapPairs[assetTo][assetFrom]`, but the mapping is configured with the swap data as `swapPairs[assetFrom][assetTo]`. This means both functions will either revert or use wrong swap data.
Recommendation: Get the swap data as `swapPairs[assetFrom][assetTo]` consistently across the code.
Severity: Medium
Resolution: The issue was resolved as recommended

VaultCore.sol

VC1. Harvest may skip taking liquidation snapshot before disabling farm mode [medium] [resolved]

The `disableFarmMode` in `VaultCore` first calls `snapshotVaultLiquidation` to make the vault snapshot before disabling the farm mode. However, if the vault was liquidated, and the position is considered unhealthy, then calling `harvest` could trigger the `disableFarmMode` but without calling the `snapshotVaultLiquidation`.
That means balance and position will not be updated correctly. Also in that case the protocol will not go to emergency pause, like it would have if the disable farm mode would have been triggered from the farm mode decision maker, which does not seem to be the intention.
Recommendation: Move the check for vault liquidation above the check of the vault position health so it will not be possible to call the `harvest` function in any case if the vault has been liquidated.
Also make sure to pause the protocol if the farm mode is disabled. Also, you might also want to disable the farm mode in case the max drop percentage was reached, like the farm mode decision maker does.
Severity: Medium
Resolution: This code was completely refactored and this issue is not relevant anymore.

VaultRegistry.sol

VR1. Salt is used improperly for deployment of vaults [low] [resolved]

The vault deployment uses the `CREATE2` method, which uses a salt to generate randomness for the address of the deployed contract. The contract defines the salt for deployment as a constant, and so the address computation for deployment, which uses the salt (always the same), deployer address (which is always the registry), and the bytecode with constructor arguments passed, will result in the same address in case the same vault data is passed for 2 deployments. This means that vaults with the same `vaultCreation.vaultData` config, but different config of for example the tokens or `bufferConfig`, will not be possible to deploy using the registry. There is no reason for such limitation.
Recommendation: Either generate a salt that is based on a nonce, or add all the parameters passed in the function for the vault initialization to be used with the salt to calculate the contract address.
Severity: Low
Resolution: The issue was resolved as recommended: the salt is now the hash of the two token addresses.