Code Assessment

of the Gearbox Smart Contracts

December 13, 2021

Produced for

Gearbox

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Gearbox Team,

First and foremost we would like to thank Gearbox Protocol for giving us the opportunity to assess the current state of their Gearbox system. This document outlines the findings, limitations, and methodology of our assessment.

The documentation and the code reviewed are of a high standard. Nevertheless, the protocol logic as well as the implementation are quite complex. Neither documentation nor specification for the `LeveragedAction` contract was provided for the audit. Even though this is contract wraps existing functionality, the specification would have been helpful in clarifying the intended behavior.

This is the final report after an iteration of reviews.

All the issues uncovered by the current review have been fixed, except for a low design issue which was only partially addressed.

The communication with your team during the audit was very good and helped to resolve arising questions quickly.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further supporting your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	3
• **Code Corrected**	3
Medium-Severity Findings	8
• **Code Corrected**	8
Low-Severity Findings	13
• **Code Corrected**	12
• **Code Partially Corrected**	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Gearbox repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	31 August 2021	9df4cd488c3209145af7897fd62bbb7b0b2319e8	Initial Core Version
2	27 September 2021	f92130695ae6eb59763190812da058fa93a59592	Core Fixes After Intermediate Report
3	7 October 2021	caee67202bd29c8f4f3583c367c7c6e2a26bcf35	Core Final Fixes
4	12 October 2021	b0fb7bf342199e31b135935a9683709a95743bb5	Leveraged Actions (LA)
5	22 October 2021	2a7a9c8cf870cd8bee4b417c8f1b4d6bac51b43e	LA Fixes for Intermediate Report
6	25 October 2021	0b825ffb2bc0f30fe47355df1bfa9719c9cf2d2f	LA Final Changes
7	13 December 2021	c922f723f1f4c92e903ac6c536dac021a5b5c5a2	Final Changes

For the solidity smart contracts, the compiler version `0.7.6` was chosen.

2.1.1 Excluded from scope

The contracts excluded from scope are the following:

- `/core/DataCompressor.sol`
- all files in `/integrations`, `/fuzzing`, `/support`, `/mocks`, `/interfaces`

In the final iteration, after the main review has already completed, pre/post condition have been added as comments for some of the main functions. These conditions are used for fuzzy testing and have not been rechecked in detail.

Furthermore, we assume that the imports of `hardhat/console.sol` and the calls to `console.log` are for development purposes only and that they will be removed in the final version of the code.

2.2 System Overview

This system overview describes the initially received version ($\boxed{\text{Version 1}}$) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added subsections for each of the changes accordingly to the versions. Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

Gearbox implements a general purpose leverage protocol for ERC-20 tokens. The system can be divided into the following parts:

2.2.1 The Credit System

It consists of three contracts, the CreditAccount, the CreditManager and the CreditFilter.

1. `CreditAccount`: It represents a leverage position and holds all the position's balances acting essentially as a wallet. The owner's access to this wallet is restricted as it contains additional funds borrowed from the pool. Interaction with external protocols using the funds of the credit account can be executed through the respective adapters.

2. `CreditManager`: This contract is responsible for managing credit accounts. Each CreditManager defines an underlying token and is connected to a liquidity pool with the same underlying token. Users can open a credit account through a credit manager, this credit account is then connected to this credit manager. The value which the account holds is valuated in the underlying. The credit manager exposes the following functionalities:

   - `openCreditAccount`: Takes a credit account from the stock of accounts and moves all the user's funds and the leverage to the account.

   - `closeCreditAccount`: It trades all the tokens the account holds to the underlying, pays back the debt, returns the surplus to the owner of the account and returns the account to the stock of accounts. On successful closure users pay a fee to the protocol which is proportional to the interest (interest fee) they paid and a fee proportional to the profits they made (success fee).

   - `repayCreditAccount`: It is similar to account closure but the user pays back the debt using their own funds and not by converting balances of the credit account. Users pay the same fees as on closure.

   - `liquidateCreditAccount`: It allows any user to liquidate an undercollateralized credit account. It functions similarly to closing and repaying a credit account but sells the tokens at the credit account at a discount in order to incentivize the liquidator.

   - `addCollateral`: Adds an amount of a supported token to the credit account.

   - `increaseBorrowAmount`: It further increases the loan taken by the user.

3. `CreditFilter`: It is responsible for enforcing the policies on the usage of the credit accounts, i.e., which tokens are allowed to be traded in the platform and which adapters connecting to external protocols are allowed to be used.

2.2.2 The Pools

The pools are used to manage the liquidity of the system. Users can lend funds to the pool and accrue interest. The funds held by the pool are then used as leverage by the users that hold credit accounts.

A pool also defines a denomination asset which is used to evaluate the pool's holdings. It exposes to the users the following functionalities:

1. `addLiquidity`: Users transfer an amount of the denomination asset to the pool and mint an amount of Diesel tokens.

2. `removeLiquidity`: The users exchange the diesel token they hold for the corresponding amount of the denomination asset. Note that redemption is not guaranteed at all times as funds may be borrowed to credit accounts.

2.2.3 The Credit Account Factory

In order to reduce costs of the deployment of the credit accounts, an account renting system is implemented. Upon opening a credit account, a free credit account contract is taken from the factory. After a position's closure, the credit account is returned to the factory. In case all the credit accounts are used, a new one is created by using the cloning paradigm.

2.2.4 Checking Collateral

The system calculates the collateralization of a position using the health factor. The health factor is essentially the ratio between a discounted value of the holding of an account and the amount that has been borrowed by the account. The discount in the value aims to prevent abrupt fluctuations in the values of the assets. As long as the health factor is greater than 1, the account is considered healthy. Otherwise, it can be liquidated.

Anyone may liquidate unhealthy credit accounts.

In order to prevent adversarial actions by the users such as stealing part of the collateral, a check is done after each action on the funds at the credit account, e.g., a trade with an external platform. This check on the collateral prevents an action from leaving a credit account undercollateralized.

However, checking the health factor is gas-heavy. In order to avoid this check after each action, Gearbox introduces fast check protection. Fast check protection is another check which limits the decrease in the collateral value. More specifically, it does not allow an action to reduce the collateral value measured in the difference of the spend and incoming assets to reduce more than a specified percentage. An additional safeguard is that after a certain number of fastchecks, a full health check has to be performed.

There are two variations of the check of the collateral. One for simple exchanges between two assets and one for exchange of multiple collaterals. The latter variant is not currently used in the reviewed system.

Note that fast check cannot cover for the edge case when the collateral is close to 1 and a non-profitable trade reduces the health factor under 1.

2.2.5 Adapters

The credit accounts can interact with external protocols via the adapters. The adapters are the only entry points which allow the aforementioned interaction. The adapters currently implemented by the system are the following:

- `UniswapV2` and `UniswapV3` which allow the credit account to trade its holdings for other assets.
- `YearnV2` which allows the credit account to deposit and withdraw assets from yearn vaults
- `CurveV1` which facilitates arbitrage with leverage on tokens which are part pools that the underlying token of the credit account also participates.

Generally, the adapters are implemented to mimic the function interface of the DeFi contract by implementing the functions with the same name and parameters. The adapters process the call on the function of the 3rd party DeFi system this adapter connects to before executing a check on the new state of the credit account e.g. using *checkCollateralChange*. This ensures that the action did not make the credit account unhealthy.

The current assumption of the adapters is that the balance of the asset sent to the external protocol will not increase and the balance of the asset received from the external protocol will not decrease.

2.2.6 Governance

The protocol is governed by the use of Gearbox Token. Users can delegate their tokens to other users or themselves to be eligible to vote. The Gearbox tokens are distributed through the `TokenDistributor` contract which was added at Version 7. This contract defines different receivers of the Gearbox Token. In particular:

- Receivers with A-Voting Power. For each such receiver, a vesting contract (`StepVesting`) is deployed which gradually unlocks the tokens. These users can use part of their unvested tokens to vote.

- Receivers with B-Voting Power. These users can use a smaller portion of their unvested tokens and are similar to A-Voting Power users otherwise.

- Receivers with 0-Voting Power. These are addresses that represent companies. They don't get any voting power except for this allowed by their vesting tokens.

- Account Miners. These are addresses which participate in the account mining. Their tokens are not vested. Users are stored in a Merkle tree and authorized by the `AccountMining`.

- The treasury. A part of the tokens is going to be stored in the treasury.

- Airdrop Testers. A part of the tokens is dedicated to the testers.

2.2.7 Trust Model

The system relies heavily on the `Configurator` role since they set the parameters of the system with few restrictions. Hence, the configurator is a role trusted by the system and is supposed to act honestly and correctly.

In general, more roles are implemented through the `ACL` which is the common authorization layer shared by the whole system. There, more roles are defined, i.e., the `pausableAdmin` and the `unpausableAdmin` who can pause and unpause the system respectively.

Moreover, the system heavily relies on the prices the Chainlink oracles provide to the system. Should the oracles behave improperly, the system can evaluate the credit accounts erroneously and allow liquidations that should not take place.

Tokens enabled for use in the system are assumed to be non-malicious ERC-20 tokens without callbacks.

Finally, the system interacts with third-party protocols, namely, `UniswapV2`, `UniswapV3`, `CurveV1` and `YearnV2`. These protocols are assumed to work correctly. Moreover, any malfunction of these protocols can seriously compromise the security and the correct behavior of the system.

Users are generally untrusted.

2.2.8 VERSION 4

This version extends the system by adding the `LeveragedActions` contract. This contracts wraps calls to the core functionality of the system, allowing the users to execute multiple core functionalities in one transaction. In particular:

- `openLong`: With the user supplying some amount of collateral `S`, opens a leveraged account for this collateral `S`. Next a swap operation is executed using the funds of the created leverage credit account and the parameters specified in the supplied `longParams` argument. Three swap adapters are currently supported, UniswapV2/V3 and Curve. The creditManager may restrict which adapters are allowed. The parameters for the swap amounts are specified by the caller. Note that the swap input parameters representing `amountIn` and `amountOutMin` respectively are expected to not include the leverage. Optionally the users can choose to deposit the resulting asset into a liquidity pool, currently only depositing all amount of the resulting asset to `YearnV2` is supported.

- `openShort(UniV2, UniV3, Curve)`: the user defines a token `L` which is traded through `UniswapV2`, `UniswapV3` or `Curve` to a token `S`. Note that the allowed swap contract must be whitelisted by the CreditManager, hence not all options may be available to the user. With the resulting balance of token `S` a leverage credit account is opened. Then, the leveraged amount of token `S` is traded for a token `L'` given the amounts specified in `longParams` using the same functionality as in `openLong`. Optionally, users can deposit the resulted amount to `YearnV2`.

- `openLP`: opens a leveraged account which is then deposited to a yield accruing protocol i.e., `YearnV2` in the current release.

2.2.9 VERSION 5

There are two important changes in this version:

- For `transferOwnership`, new restrictions have been applied whenever the sender is an address unknown to the system or the receiver is an address known to the system. In this case, in order for the receiver can get a Credit Account only after they have given an allowance. In other words, `transferOwnership` will fail if the receiver has not explicitly given their consent to receive an account from the sender.

- The success fees have been removed.

3   Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- `Security` : Related to vulnerabilities that could be exploited by malicious actors
- `Design` : Architectural shortcomings and design inefficiencies
- `Correctness` : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

`Critical`-Severity Findings	0
`High`-Severity Findings	0
`Medium`-Severity Findings	0
`Low`-Severity Findings	1

- Missing Sanity Checks `Code Partially Corrected`

5.1 Missing Sanity Checks

`Design` `Low` `Version 4` `Code Partially Corrected`

When opening a short position in the Gearbox system by calling `shortOpenUniV2`, the user must provide multiple parameters. These parameters are not sanitized, thus arbitrary behavior may occur. More specifically it is never checked that `path[path.length - 1] == collateral` and `collateral == longParams.path[0]`.

The `lpInterface` and `lpContract` in the struct `LongParameters` used in `_openLong` are not checked to match. Similarly, an arbitrary router can be passed as `shortSwapContract` as long as there is an adapter for it. Note that this is currently not an issue since different adapters/routers do not share the same interface and the transaction would revert. However, the addition of more adapters in the future might require some kind of sanity check.

Code partially corrected:

`shortOpenUniV2` now features an additional check ensuring that the token out of the exchange using `shortSwapContract` is the collateral. Similar checks have been added to `openShortUniV3` and `openShortCurve`.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical -Severity Findings	0

High -Severity Findings	3

- Incorrect Arguments in checkCollateralChange `Code Corrected`
- Non-Accessible Credit Accounts `Code Corrected`
- Retain Ownership of Credit Account `Code Corrected`

Medium -Severity Findings	8

- DoS of LeverageActions `Code Corrected`
- Incorrect params.amountOutMinimum `Code Corrected`
- Contracts Implement Proxy Pattern `Code Corrected`
- Trust Model of External Adapters `Specification Changed` `Code Corrected`
- Users Can Avoid Paying Fees On Closure `Code Corrected`
- Wrong Approval To Pool `Code Corrected`
- maxAmount Can Be Circumvented `Code Corrected`
- takeOut May Break the Account List `Code Corrected`

Low -Severity Findings	12

- Discrepancy Between openShortUniV2 and openShortUniV3 `Code Corrected`
- Use of transfer `Code Corrected`
- Rounding Errors `Code Corrected`
- Head Cannot Be Taken Out `Code Corrected`
- Pointers Not Updated On takeOut `Code Corrected`
- Redundant Multiplication `Code Corrected`
- Storage Optimizations `Code Corrected`
- Taking Out the First-Ever Created Account `Code Corrected`
- allowToken Can Be Blocked `Code Corrected`
- cancelAllowance Cannot Be Called `Code Corrected`
- connectCreditManager Access Control `Code Corrected`
- rayMul and rayDiv Are Used With No Ray Values `Code Corrected`

6.1 Incorrect Arguments in `checkCollateralChange`

`Correctness` `High` `Version 2` `Code Corrected`

In `YearnV2.withdraw(uint256, address, uint256)`, the `checkCollateralChange` is called with wrong arguments. Particularly, the following snippet is used:

```
creditFilter.checkCollateralChange(
    creditAccount,
    token,
    yVault,
    balanceInBefore.sub(IERC20(yVault).balanceOf(creditAccount)),
    balanceOutBefore.add(IERC20(token).balanceOf(creditAccount))
);
```

Note that `token` is the `tokenOut` in this particular case, we convert `yVault` tokens to the underlyings and `yVault` is the `tokenIn`. This error later results in querying the oracles with wrong balances.

Code Corrected:

The arguments are now passed correctly to `checkCollateralChange`.

While the final round of the review was ongoing Gearbox Protocol informed us of an issue in the new implementations of the adapters. The adapters were calculating the delta of the incorrectly and hence were passing wrong parameters to `checkCollateralChange`. The issue has been fixed.

6.2 Non-Accessible Credit Accounts

`Security` `High` `Version 1` `Code Corrected`

The `transferAccountOwnership` function of a CreditManager contract allows the owner of a credit account to transfer it onwards to a new owner. Per CreditManager an address is only allowed to hold one credit account. `trasferAccountOwnerhip()`. However, there is no check on whether the recipient already holds a credit account at this CreditManager contract and simply overwrites the entry for the credit account of the recipient. Hence a credit account which holds funds can become non-accessible and its funds will be trapped.

Code corrected:

In the updated code the `transferAccountOwnership` function no longer overwrites an existing credit account entry of the recipient, hence the issue no longer exists.

6.3 Retain Ownership of Credit Account

`Security` `High` `Version 1` `Code Corrected`

In Gearbox, Credit Accounts are reused after they have been returned to the factory. Due to a reentrancy issue, account ownership can be retained and after the next user got this credit account assigned, the previous owner may access its funds belonging to the new owner.

Function `transferAccountOwnership` does not feature the non `nonReentrant` modifier and hence can be executed during another operation. Consider the follwowing scenario:

Alice owns a healthy credit account `0xA` which holds some WETH balance.

1. Alice prepares a contract that allows her to execute all necessary actions. As a first step, the credit account ownership is transferred to this contract.

2. The credit account is repaid using `repayCreditAccount` specifying the contract as `to` address. This transfers all assets to the provided `to` address. Notably the WETH asset is unwraped into Ether, the Ether is transferred in a call to the reciepient's address `to`. This call executes code at the contract.

3. At the specified `to` address a contract exists. This contract transfers the ownership of the credit account onwards to another address (`newAddress`) Alice controls. This means that `creditAccounts[newAddress]` will point to the credit account

4. The closure of the credit account continues as normal. All assets are transferred to address `to`, the debt is repaid to the pool and the credit account is returned to the AccountFactory.

5. Next `delete creditAccounts[borrower];` is executed, this should delete the assignment of this credit account to the `borrower`. However, as we already transferred the ownership from `borrower` which is the contract address back to Alice, `creditAccounts[borrower]` contains no entry at this point and deleting it has no effect.

At the end of this sequence, the credit account has been returned to the AccountFactory but the entry `creditAccounts[newAddress]` in this CreditManager still points to this account.

The next time this `CreditAccount` is reused at the **same** CreditManager by a new user, due to the entry in `creditAccounts` Alice will still have access to this account and can collect its funds by e.g. closing or repaying the account.

Code corrected:

`transferAccountOwnership()` now features the `nonReentrant` modifier. Hence, the reentrancy issue described is no longer possible.

6.4 DoS of `LeverageActions`

Design Medium Version 4 Code Corrected

`LeveragedActions` can be blocked completely or for specific collaterals only in different ways:

1. When opening an account the credit manager will check if `onBehalfOf` already has an account. In case a malicious user has already transferred the ownership of a credit account to the LeverageActions contract then the CreditManager will fail to open a new one:

```
function openCreditAccount(
    ...
    require(
        !hasOpenedCreditAccount(onBehalfOf) && onBehalfOf != address(0),
        Errors.CM_ZERO_ADDRESS_OR_USER_HAVE_ALREADY_OPEN_CREDIT_ACCOUNT
    ); // T:[CM-3]
    ...
```

2. Although this is more a theoretical attack, assume a credit manager which prohibits the user to invest more that `A` amount of tokens. A malicious user sends to the the contract `A + 1` tokens. When the contract will try to open a leveraged position it will do so using the total balance of the token it holds. If this amount is greater than the allowed one the account opening will block. The snippets which dictate the above behavior are the following:

```
LeverageActions:

function _openLong(LongParameters calldata longParams, uint256 referralCode){
```

```
    ...
    uint256 amount = IERC20(collateral).balanceOf(address(this)); // M:[LA-1]
    ...
}

CreditManager:

function openCreditAccount(
    ...
    require(
        amount >= minAmount &&
            amount <= maxAmount &&
            leverageFactor > 0 &&
            leverageFactor <= maxLeverageFactor,
        Errors.CM_INCORRECT_PARAMS
    ); // T:[CM-2]
    ...
}
```

Code corrected:

For the case #1, an allowance system was implemented for the transfer of credit account. In order to get a credit account transferred, the receiver needs to pre-approve the sender of the credit account. Hence one can no longer transfer a credit account to the LeveragedAction contract and the issue no longer exists.

To mitigate case #2 the LeveragedActions contract now uses the actual balance difference.

Moreover, Gearbox Protocol pointed out a third way to use the attack described above. Specifically, a user can open an account on behalf of the LeverageAccount contract which would result in a Denial-of-Service for the LeverageAction contract. The issue has been resolved by also restricting the address on behalf of which the credit account is opened.

6.5 Incorrect `params.amountOutMinimum`

`Correctness` `Medium` `Version 4` `Code Corrected`

The parameter `params.amountOutMinimum` passed to the call to the UniswapV3 adapter in `_openLong()` is calculated incorrectly and does not include the leverage.

`_openLong` executes a swap using the funds of the opened leveraged account given the swap parameters in `longParams`. The relevant parameters for the swap are in `bytes swapCalldata` which are first extracted and prepared for the call to the swap contract. Note however the parameters representing `amountIn` and `amountOutMinimum` extracted from `swapCalldata` do not include the leverage, hence the actual values for the swap have to be calculated:

```
else if (longParams.swapInterface == Constants.UNISWAP_V3) {
    ISwapRouter.ExactInputParams memory params = abi.decode(
        longParams.swapCalldata,
        (ISwapRouter.ExactInputParams)
    );

    params.amountIn = leveragedAmount;
    params.amountOutMinimum = params
    .amountOutMinimum
```

```
    .mul(leveragedAmount)
    .div(params.amountIn);
    ISwapRouter(adapter).exactInput(params);
    (, asset) = _extractTokensUniV3(params.path);
}
```

First `params.amountIn` is overwritten with `leveragedAmount`. Next `params.amountOutMinimum` is calculated, this calculation uses `params.amountIn` which is equal to `leveragedAmount` at this point.

Hence the calculation: `params.amountOutMinimum.mul(leveragedAmount).div(params.amountIn);` actually is `params.amountOutMinimum.mul(leveragedAmount).div(leveragedAmount);` which simplifies to `params.amountOutMinimum`.

The leverage is not included in `params.amountOutMinimum`.

Code corrected:

The calculation of the leveraged value for `params.amountOutMinimum` is now done correctly using the unchanged value of the decoded `params.amountIn`. `params.amountIn` is only set to `leveragedAmount` afterwards.

6.6 Contracts Implement Proxy Pattern

`Security` `Medium` `Version 1` `Code Corrected`

All adapters and the YearnPriceFeed contract inherit from OpenZeppelin's abstract Proxy contract and implement an `_implementation` function pointing to the address of the 3rd party system contract the adapter connects to. However, this proxy functionality is not needed nor used. The intended functionality of the adapter is implemented in functions inside the adapter contract itself.

Inheriting the proxy contract, however, has serious consequences. Calls to non-existing functions in the contract execute the fallback function, which is implemented by the inherited proxy. This function forwards the call by delegate-calling into the implementation contract. During a delegate-call, the code at the target is executed in the context of the caller. Notably, it is read from and written to the storage of the caller, the adapter contract. This can have an adverse effect on the stored variables of the adapter contract. For example the stored values for the creditManager or the creditFilter.

Code corrected:

The adapter contracts were rewritten and the proxy pattern was removed.

6.7 Trust Model of External Adapters

`Design` `Medium` `Version 1` `Specification Changed` `Code Corrected`

The trust model for the external adapters has not been properly specified. Moreover, all four available adapters behave differently and the assumptions these adapters rely on have not been documented.

After the action on the external system which is invoked by an adapter, there is a check on the collateral of the credit account. All currently available adapters use the following function which takes the following parameters:

```
function checkCollateralChange(
    address creditAccount,
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint256 amountOut
)
```

The concern is about what is passed as amount especially for the spent asset. It is vital that these amounts represent the actual state of the credit accounts holding or the check may be circumvented.

Some adapters rely on the values returned by the 3rd party system, some query the actual balance.

While querying the actual balance for the assets involved in the action is the safest option, it may be expensive in terms of gas. However note that in the current implementation of the EVM (London hardfork), repeated access to the same contract/storage location got significantly cheaper the overhead in terms of gas may not be that big.

Using values returned by the call to the third-party contract may be an option if the third-party contract is fully trusted to do so correctly. Similarly, this holds for input parameters. This critical part should be documented and assessed thoroughly. In case of doubts/uncertainties, it may be safer to query the balances and calculate the delta of the balances and use this.

Regarding the `YearnAdapter`, it can be inspected and documented: Querying the balances could be avoided since both `Vault.deposit` and `Vault.withdraw` [https://github.com/yearn/yearn-vaults/blob/main/contracts/Vault.vy] return the change in the balance of the tokens of interest. However, the current YearnAdapter does not do this but queries the balance and calculates the delta.

The UniswapV3 Adapter relies on the returned values by the 3rd party system. However, there is no documentation why this assumption holds.

Specification changed and code corrected:

A pattern of how all adapters should be built has been created. All existing adapters have been rewritten to adhere to this pattern: The balance is queried before and after the action and the difference is used for the check of the collateral change.

Note that due to the existing token allowances for the adapters from the credit accounts these checks are not 100% failsafe. It is vital that the 3rd party system is fully trusted to not transfer any other tokens of the credit account. The system performs the fast check only for the tokens passed as arguments to the check. Any other change in balance will be ignored.

6.8 Users Can Avoid Paying Fees On Closure

`Design` `Medium` `Version 1` `Code Corrected`

On account closure, all the assets held by the account are converted to the underlying token through `defaultSwapContract` which is set to be UniswapV2. For this conversion, the user defines a path of tokens to the underlying. This path can contain arbitrary tokens, tokens even controlled by the user. A check in `_closeCreditAccountImpl` assures that the closure of a credit account will not lead to losses for the protocol i.e., `require(loss <= 1)`. On the closure of an account users are supposed to return to pool the amount they borrowed, the interest accrued for that amount and an extra amount for fees namely, `feeSuccess` and `feeInterest`. It is important to note that if the funds do not suffice `totalFunds < amountToPool` then only the borrowed amount with the interest accrued is returned and no fees are required to be paid. This means that draining a credit account to the point that does not make losses can allow a user to avoid paying fees to the protocol.

Code Corrected:

A new check has been introduced which requires that `remainingFunds > 0`. This way it is guaranteed that the user has paid their fees. Due to this requirement, a closure that does not result in fee payout will be reverted. Hence, the only option for the users will be to repay.

6.9 Wrong Approval To Pool

Design Medium Version 1 Code Corrected

While the review was ongoing Gearbox Protocol informed us about this issue independently in parallel.

In the `WETHGateway.repayCreditAccountETH` an approval is given to the pool:

```
_checkAllowance(pool, amount); // T: [WG-11]
```

However, this approval is wrong and should be given to the credit manager who performs the transfer from the `WETHGateway` to the pool.

Code corrected:

The code has been corrected in a further commit and the allowance is now given to the CreditManager instead of the pool in order for the credit manager to be able to transfer the tokens from the user to the pool.

6.10 `maxAmount` Can Be Circumvented

Design Medium Version 1 Code Corrected

When opening a credit account, a check of the amount invested is performed:

```
require(
    amount >= minAmount && amount <= maxAmount,
    Errors.CM_INCORRECT_AMOUNT
);
```

By limiting the amount originally invested, one can limit the amount of leverage that can be borrowed by the pool. However, this limitation can be circumvented as follows:

1. The user opens an account with an allowed account.
2. She calls `CreditManager.addCollateral`.
3. She calls `increaseBorrowedAmount`.

Note, that `addCollateral` does not perform any checks and `increaseBorrowedAmount` only checks that the borrowed amount does not turn the account unhealthy.

Code Corrected:

The implementation has been extended to prevent increasing the borrowed amount more than the predetermined maximum:

```
require(
    borrowedAmount.add(amount) <
        maxAmount.mul(maxLeverageFactor).div(
            Constants.LEVERAGE_DECIMALS
        ),
    Errors.CM_INCORRECT_AMOUNT
);
```

6.11 `takeOut` May Break the Account List

`Design` `Medium` `Version 1` `Code Corrected`

The configurator can take out an account by calling `AccountFactory.takeOut()`. During account removal, there is no check whether this is the tail nor is the tail updated in case this account is taken out. Should the tail account be taken out this is problematic:

New accounts added will not be connected to the original list, hence they cannot be taken using `takeCreditAccount()` which takes the head of the original list.

Similarly, returned accounts will be added to the list after the removed `tail` account which no longer exists in the list. Again, the connection to the original list starting at `head` is interrupted and these accounts cannot be used anymore.

Code Corrected:

The implementation has been extended to correctly update `tail` when the last account is taken out.

6.12 Discrepancy Between `openShortUniV2` and `openShortUniV3`

`Design` `Low` `Version 4` `Code Corrected`

`LeverageAction.openShortUniV2` sets the deadline for the short swap to the current block timestamp:

```
bytes memory data = abi.encodeWithSelector(
    bytes4(0x38ed1739), // "swapExactTokensForTokens(uint256,uint256,address[],address,uint256)",
    amountIn,
    amountOutMin,
    path,
    address(this),
    block.timestamp
); // M:[LA-5]
```

This way the call cannot fail due to a passed deadline. On the other hand, `LeverageAction.openShortUniV3` lets users define the deadline themselves. This means that a transaction that takes long to be included into a block might fail.

Code corrected:

The code of `openShortUniV2` was changed and now uses the user-specified parameter `deadline` instead of `block.timestamp`. It's the caller's responsibility to specify a proper deadline. With this change, the behavior of the functions for UniV2 and V3 are now consistent.

6.13 Use of `transfer`

`Design` `Low` `Version 4` `Code Corrected`

`_returnTokenOrUnwrapWETH` uses `transfer` instead of `safeTransfer` for transferring tokens. This call will fail for tokens which do not adhere to the `ERC20` interface e.g., `USDT`.

Code corrected:

The code was changed to use `safeTransfer`.

6.14 Rounding Errors

`Design` `Low` `Version 2` `Code Corrected`

In `CreditManager.increaseBorrowedAmount` the following check is performed:

```
require(
    borrowedAmount.add(amount) <
        maxAmount.mul(maxLeverageFactor).div(
            Constants.LEVERAGE_DECIMALS
        ),
    Errors.CM_INCORRECT_AMOUNT
);
```

This check includes a division with `Constants.LEVERAGE_DECIMALS` which results in a rounding error. This error can be avoided, if one multiplies the left side of the inequality with the same value instead.

In the following snippet of `PoolService.expectedLiquidity` a division before multiplication takes place:

```
uint256 interestAccrued = totalBorrowed
.mul(borrowAPY_RAY)
.div(Constants.RAY)
.mul(timeDifference)
.div(Constants.SECONDS_PER_YEAR);
```

Division before multiplication can result in rounding errors. In this particular case, the `interestAccrued` will be smaller.

Code Corrected:

Regarding the first issue, the division has been replaced with a multiplication. Regarding the second one, the order of operations has changed and the multiplications take place first.

6.15 Head Cannot Be Taken Out

Design Low Version 1 Code Corrected

Calling `AccountFactory.takeOut` requires to pass the previous account of the one to be deleted (`prev`). This means that the head credit account of the list cannot be taken out since there is no `prev` defined for it.

Code Corrected:

The implementation has been extended to handle the removal of the head.

6.16 Pointers Not Updated On `takeOut`

Design Low Version 1 Code Corrected

A credit account can be taken out of the system by the configurator using function `AccountFactory.takeOut`. Under normal circumstances this account cannot be accessed again by the function. However, consider the following scenario:

1. The controller removes the head account (`A1`). In this case, the head is just updated to the second account (`A2`). Note that at the removal of the head, the pointers of the head account `_nextCreditAccount[head]` is not reset.

2. Later `A2`, the current head is also removed.

3. This means that the controller can take out `A2` again by calling `takeOut(A1, A2)` and connect it to a new `to` address.

The reason for the above is that `_nextCreditAccount[A1]` is not updated upon removal and still points to `A2` which has also been removed. The check

```
require(
        _nextCreditAccount[prev] == creditAccount,
        Errors.AF_CREDIT_ACCOUNT_NOT_IN_STOCK
    );
```

is still satisfied despite the accounts being no longer part of the system.

Code Corrected:

The pointers are now updated correctly.

6.17 Redundant Multiplication

Design Low Version 1 Code Corrected

In `PoolService.removeLiquidity` a part of the amount requested by the user is sent back to them determined by `withdrawMultiplier` and an amount determined by the `withdrawFee` is sent to the treasury. By construction we know that `withdrawMultiplier + withdrawFee == PERCENTAGE_FACTOR`. These two amounts should add up to `underlyingTokensAmount`. Hence, there is no need to perform two safe multiplications with both `withdrawFee` and `withdrawMultiplier` and the following multiplication is redundant:

```
IERC20(underlyingToken).safeTransfer(
    ...
    underlyingTokensAmount.percentMul(withdrawFee)
); // T:[PS-3, 34]
```

Code Corrected:

The issue has been resolved. In the current implementation, only one multiplication takes place. The amount sent to the treasury is now calculated by subtracting `amountSent` from `underlyingTokensAmount`.

6.18 Storage Optimizations

Design Low Version 1 Code Corrected

There are various small optimizations that can be applied to the contracts of the system to improve gas efficiency:

1. Storage variable can be declared as constants: In GearToken contract `totalSupply` can be declared as constant.

2. Some functions can be declared as `external`:

- `AccountFactory.countCreditAccountsInStock()`
- `CreditFilter.checkCollateralChange(address,address,address,uint256,uint256)`
- `CreditFilter.allowedContractsCount()`
- `CreditFilter.allowedContracts(uint256)`
- `GearToken.delegate(address)`
- `GearToken.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32)`
- `GearToken.getPriorVotes(address,uint256)`

3. Dead code which can be removed:

- `BytesLib.slice(bytes,uint256,uint256)`
- `BytesLib.toUint24(bytes,uint256)`

Code Corrected:

Issues 1. and 2. have been resolved. Regarding 3., the client states:

> BytesLib functions are used in support contracts which are not in the scope

6.19 Taking Out the First-Ever Created Account

Correctness Low Version 1 Code Corrected

The configurator can call `AccountFactory.takeOut` to remove an account completely and connect it to an address of their choice. To do so, they provide the address of the account to be removed and the address of the previous account in the list of the available accounts. Let us consider the addition of the

first-ever created account. The account is added during the deployment of the `AccountFactory` i.e., when the constructor is invoked. At this point, both the `head` and the `tail` are `0`. This means that in the following snippet, it holds `_nextCreditAccount[0] == clonedAccount`.

```
function addCreditAccount() public {
    ...

    _nextCreditAccount[tail] = clonedAccount; // T:[AF-2]

    ...
```

Note that `_nextCreditAccount[0]` is never updated. This means that there is always a pointer at `0` to the first-ever created account. If the configurator calls `takeOut` with `prev == 0x0` and `creditAccount` the first ever created account they can control it even though the account might be in use at the time of the call. In other words, there is always a pointer to the first ever created account even if the account is not in stock. The case above makes the following check in `AccountFactory.takeOut` and the error message emitted imprecise:

```
require(
    _nextCreditAccount[prev] == creditAccount,
    Errors.AF_CREDIT_ACCOUNT_NOT_IN_STOCK
); // T:[AF-15]
```

The check whether the account is in stock doesn't work as expected in the scenario described above.

Code Correct:

The pointer of `_nextCreditAccount[0]` now points to `address(0)` and not the first-ever created account.

6.20 `allowToken` Can Be Blocked

Design Low Version 1 Code Corrected

The purpose of `creditFilter.allowToken` is twofold. On one hand, it allows the system to use new tokens. On the other hand, in the case of an already registered token, it allows updating the liquidation threshold for this token.

Due to the bitmask optimization used, the following check assures that no more than 256 different tokens can be tracked by the system.

```
require(allowedTokens.length < 256, ...);
```

However, in the unlikely case of 256 registered tokens, the liquidation threshold cannot be updated anymore since the above check will fail, leading the transaction to revert.

Code Corrected:

The code has been corrected. The requirement will be satisfied when the function is called with a token for which `tokenMasksMap[token] > 0` as shown in the following in snippet:

```
require(
    tokenMasksMap[token] > 0 || allowedTokens.length < 256, ...
);
```

6.21 `cancelAllowance` Cannot Be Called

`Design` `Low` `Version 1` `Code Corrected`

When an account is closed, it is returned to the factory. It is important to note, however, that the allowances the account has given to other addresses remain in place. This can be dangerous in case of malfunctioning approved contracts. In order to mitigate this risk, the `configurator` is allowed to reduce or remove the allowances. This functionality is implemented by `CreditManager.cancelAllowance`. This function is supposed to be called by the factory. However, no function that calls `cancelAllowance` is implemented, thus the allowance cannot be revoked.

Code Corrected:

The code has been corrected. In the current implementation the configurator can call `AccountFactory.cancelAllowance` which then calls `CreditAccount.cancelAllowance`.

6.22 `connectCreditManager` Access Control

`Design` `Low` `Version 1` `Code Corrected`

The `CreditFilter.connectCreditManager` function does not implement proper access control. The first caller to this function can set CreditManager to his address. This does not pose threat to the system but could lead to wasted deployments of the Credit Filter.

Code Corrected:

The code has been fixed, now only the configurator is allowed to set the `creditManager` for the filter.

6.23 `rayMul` and `rayDiv` Are Used With No Ray Values

`Correctness` `Low` `Version 1` `Code Corrected`

`PoolService.expectedLiquidity()` performs a multiplication using `rayMul` passing `totalBorrowed` as a parameter. However `totalBorrowed` is not in `RAY` but in the decimals of the underlying token.

```
uint256 interestAccrued = totalBorrowed.rayMul(
    borrowAPY_RAY.mul(timeDifference).div(Constants.SECONDS_PER_YEAR)
); // T:[GM-1]
```

This contradicts the specification for `rayMul` which states the following:

```
* @dev Multiplies two ray, rounding half up to the nearest ray
```

Similarly this applies for `fromDiesel()`. Additionally `getDieselRate_RAY()` uses and `toDiesel()` use `rayDiv` which is annotated with:

```
* @dev Divides two ray, rounding half up to the nearest ray
```

Code corrected:

`rayMul` and `rayDiv` are now correctly used.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Blocking `updateContributors`

Note **Version 7**

`TokenDistributor.updateContributors` can block. The function makes use of `TokenDistributor.updateVesting` for each holder. Consider the following scenario: A receiver `RA` of the Vesting contract calls `setReceiver` to an address `RB` which is a receiver of another contract. Then for `updateVesting(RA)`, it holds `vestingContracts[RB].contractAddress != 0` which makes the transaction revert. This leads the execution of updateContributors to revert as well. Note that users do not have an incentive to change the receiver address to another's receiver address. Moreover, the new receiver can change again the address to another public address they control. This would unblock the execution of `TokenDistributor.updateContributors`. However, it is up to the specific user to address the issue.

This just affects the `updateContributors` function which attempts to update all holders. The unaffected holders can always be updated individually through `updateVesting()`.

7.2 Handling Of Reward Tokens

Note **Version 1**

Users of the Gearbox system are allowed to trade through specific adapters. Moreover, the credit accounts are only enabled to access the balance of the enabled tokens which are specified by the governance. However, there might be the case where one of the allowed tokens accrues rewards in another token which is not part of the enabled tokens. Currently, users can only collect their rewards by repaying their accounts and receive the tokens which accrue the rewards.

Furthermore, rewards may be accrued by the credit account address e.g., due to a user interacting with a certain third-party system. Such a reward may be only claimable in the future, notably e.g., after a credit account user returned his account to the factory. Such a reward may be claimable by the next user of this credit account.

7.3 Liquidity Removal Not Always Possible

Note **Version 1**

Users can remove liquidity they have offered to the pool by calling `PoolService.removeLiquidity`. During this call, a transfer is performed from the pool to the `msg.sender` with the requested amount. It is important to be aware that in case of high utilization of the pool, the amount requested might not be available since it is used as leverage in some positions.

7.4 Oracles Do Not Handle Stale Prices

Note **Version 1**

The Gearbox system relies on chainlink oracles to derive the value of the assets a credit account holds. The chainlink interface allows the consumers of the data to know whether a price returned is stale or not based on the timestamps https://docs.chain.link/docs/price-feeds-api-reference/#latestrounddata. However, Gearbox does not take advantage of these timestamps meaning that stale data could be used by the system.

7.5 Price Feeds Cannot Be Updated

Note **Version 1**

A price feed can be added by the configurator of the system by calling `PriceOracle.addPriceFeed`. The logic of the addition is implemented inside an if statement with the following condition:

```
if (priceFeeds[token] == address(0)) {
```

This means that if the price feed for a token `T` is already defined i.e., `priceFeeds[T] != 0` then it cannot be updated. This becomes important especially when it comes to custom price feed such as the yearn price feed which might require an upgrade at some point.

7.6 Special ERC-20 Token Behavior May Be Problematic

Note **Version 1**

Some ERC-20 tokens have transfer fees. Supporting such tokens may lead to accounting errors as the actual amount received after a transfer may not match the expected amount, e.g. when funds are repaid to the pool.

Furthermore, note that the `_convertAllAssetsToUnderlying()` used during the closure of a credit account uses UniswapV2's `swapExactTokensForTokens` function which does not support token with transfer fees.

In general, when adding tokens to the system they should be carefully inspected for any special behavior such as hooks. If any special behavior is detected, the impact on the system should be evaluated carefully.

7.7 Users Can Turn Their Account Liquidatable Inadvertently

Note **Version 1**

Gearbox uses fast check and health factor in order to prevent users from draining funds that should be returned back to the pool.

However an unaware user may turn his account into a liquidatable state inadvertently. Consider the following scenario:

Assume that a healthy account holds only token `A` with value `V_A` (in the underlying token) and owes amount `B`. The health factor of the account is `H_f = V_A * LT_A / B`.

Now, this user trades the balance of `A` to token `C`, which is worth slightly when evaluated in the underlying asset. After the trade through the adapter is completed, the check on the collateral takes place. Let's assume we're eligible for the fast check and this passes as the value in terms of the underlying has increased.

However, it could be that the liquidation threshold of token A and token C are different, e.g. `LT_C << LT_A`. This means that the health factor `H_f' = V_C * LT_C / B` may become less than 1 after the trade even though the value of the holdings has not been decreased.