PUBLIC

Code Assessment

of the Gearbox Auction Smart Contracts

December 12, 2022

Produced for

Gearbox

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Gearbox team,

Thank you for trusting us to help GEARBOX with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Gearbox Auction according to Scope to support you in forming an opinion on their security risks.

GEARBOX implements a liquidity bootstrapping contract for a GEAR / ETH Curve crypto pool. The funding is raised in consecutive stages, after which the contract acts as a doorway to the Curve pool for a limited time in which GEAR sellers are paying a premium.

The most critical subjects covered in our audit are functional correctness and safety of the interactions with the underlying pool. Additionally, we focused on front-running possibilities and gas efficiency. We did not find any critical problems in the aforementioned categories. All raised issues have been fixed accordingly.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

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	0
Medium-Severity Findings	0
Low-Severity Findings	4
• Code Corrected	4

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 Auction repository `src` folder 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	05 Dec 2022	9951434beee4d3de3e3d0155c3b663fd387c6366	Initial Version
2	10 Dec 2022	855519f061e717b4f0fd455030fb014e88d6c761	Fixed Version 1

For the solidity smart contracts, the compiler version `^0.8.10` were chosen.

2.1.1 Excluded from scope

All files (including tests) outside of the `src` folder of the repository.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

The reviewed smart contracts allow GEAR token holders to bootstrap a Curve exchange with early liquidity and, later, to trade GEAR tokens vs. ETH. If certain conditions are met, transfer restrictions currently imposed on all GEAR holders (except the treasury and a miner address) are lifted after a pre-determined timeframe.

The repository has two main files. `constants.sol` holds the constant parameters like the GEAR token address, the Curve deployer address, token limits for the deposit phases and Curve's trading curve parameters. The main contract `GearLiquidityBootstrapper.sol` is composed of the following stages:

- `INITIALIZED`: The stage at the beginning of the contract's lifetime.
- `GEAR_DEPOSIT`: The stage in which users are allowed to deposit GEAR tokens into the contract. If `gearMinAmount` has been reached after a certain time, the contract progresses to the next stage. Otherwise, it progresses to `FAILED`.
- `ETH_DEPOSIT`: The stage in which users are allowed to deposit ETH into the contract. If `ethMinAmount` has been reached after a certain time, the contract deploys a new GEAR / ETH Curve pool, bootstraps it with the deposited liquidity and progresses to the next stage. Otherwise, it progresses to `FAILED`.
- `FAIR_TRADING`: The stage in which users are allowed to trade on the newly created Curve pool. Since the GEAR token still does not permit transfers, directly trading on the pool is not possible.

After a certain time, the contract progresses to the `FINISHED` stage and enables transfers in the GEAR token.

- `FINISHED`: The stage in which users who deposited in the beginning can claim the LP tokens of the Curve pool.
- `FAILED`: The stage in which the contract has failed to reach its purpose. The community will have to decide on a different strategy at this point.

`GearLiquidityBootstrapper` provides the following logic:

- `commitGEAR` is only callable in the `GEAR_DEPOSIT` stage. It allows GEAR token owners to deposit GEAR tokens (up to `gearMaxAmount`) in the contract.
- `commitETH` is only callable in the `ETH_DEPOSIT` stage. It allows users to deposit ETH (up to `ethMaxAmount`) in the contract.
- `sellGEAR` can be called in the `FAIR_TRADING` stage and allows users to sell their GEAR tokens on the newly created Curve pool. Users selling very early are paying a penalty which decreases over time.
- `buyGEAR` allows users to buy GEAR tokens for ETH in the `FAIR_TRADING` stage. Buying GEAR tokens is done without any discounts at the full amount.
- `claimLP` can be used by users in the `FINISHED` stage to claim their LP token share for their deposits.
- `retrieveShearedGEAR` is a function also only callable in the `FINISHED` stage. It allows the contract owner to claim the remaining GEAR token balance of the contract.
- `advanceStage` is a time and limit dependent function. It advances through the contract's stages and is called with every function of the contract. In case the min or max amounts are not collected or in case of incorrect time transitions, this function will let the bootstrap process fail.
- If the bootstrap process failed, `retrieveGEAR` and `retrieveETH` allow users to withdraw their deposited ETH or GEAR tokens.
- `takeGEARManagerBack` gives the DAO the ability to claim back the ownership of the GEAR token contract in case the bootstrap process failed.

The Curve pool deployed at the beginning of the `FAIR_TRADING` stage will be bootstrapped with a certain range of liquidity.

2.2.1 Trust model

- The `owner` role (which is set to the `GEARBOX_TREASURY` multi-sig address on deployment) of the Bootstrap contract has to be fully trusted. It can call a function `execute` which performs an arbitrary function call from the contract.

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 the 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 defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. 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 that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those 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 severity. 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:

- 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	0

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	0
Medium-Severity Findings	0
Low-Severity Findings	4

- Advance to Wrong Stage Code Corrected
- ETH Transfer Code Corrected
- Gas Optimizations Code Corrected
- Shearing Percent Getter Wrong Before Fair Trading Stage Code Corrected

6.1 Advance to Wrong Stage

Design Low Version 1 Code Corrected

When all deposits are done and meet the minimum requirements but no trades happen, consequently, the stage is not advanced and the stage `FAIR_TRADING` would never be active. If `claimLP` is called after `fairTradingEnd` it will not be possible to claim because the stage is still in `ETH_DEPOSIT` instead of `FAIR_TRADING` . `advanceStage` needs to be called first to allow claiming the LP tokens in this case to put the stage into `FAIR_TRADING` and allow the stage progression to be finished in the next iteration.

Code corrected:

If `fairTradingEnd` is reached in the `ETH_DEPOSIT` stage, the next call will automatically advance the stage to `FINISHED`.

6.2 ETH Transfer

Design Low Version 1 Code Corrected

The contract uses the native `transfer` function to transfer ETH to users. As this function can only use up to 2,300 gas, users using contracts (e.g., Gnosis Safe) to transfer ETH to the contract might be in for a surprise when the `FAILED` stage is reached and they want to call `retrieveETH`: The transfer won't succeed. In this case, the ETH of these users has to be transferred manually by the owner using `execute`.

Code corrected:

ETH are now transferred using low level calls.

6.3 Gas Optimizations

Design Low Version 1 Code Corrected

The following parts of the contracts could be optimized for gas efficiency:

- `_advanceStage` redundantly checks for the condition if the `block.timestamp` has exceeded the `fairTradingStart`.
- `_advanceStage` performs multiple, redundant storage loads of `stage`.
- `_advanceStage` sets the stage to `ETH_DEPOSIT` and then immediately sets it to `FAILED`.
- GEAR token transfers are conducted using `safeTransferFrom`. This overhead is not needed as the GEAR token's transfer functions are reverting by default.
- `_getCurrentMinMaxAmounts` redundantly loads `totalGearCommitted` and `totalEthCommitted` from storage multiple times.
- `getPendingLPAmount` redundantly loads `totalLPTokens` from storage multiple times.
- `_commitETH` redundantly loads `totalEthCommitted` from storage multiple times.
- `commitGEAR` redundantly loads `totalGearCommitted` from storage multiple times.
- `_depositToPool` redundantly loads `curvePool`, `totalGearCommitted` and `totalEthCommitted` from storage multiple times.
- `foundry.toml` does not contain settings for the optimizer.

Code corrected:

All listed gas optimizations have been integrated, except for the following (minor) points:

- `foundry.toml` has not been updated with optimizer settings.
- `_depositToPool` still loads `curvePool` two times from storage.

6.4 Shearing Percent Getter Wrong Before Fair Trading Stage

Correctness Low Version 1 Code Corrected

If `getCurrentShearingPct` is called before the `fairTradingStart` timestamp is reached, it will display more than the maximum shearing percentage.

Code corrected:

`getCurrentShearingPct` now returns `shearingPctStart` before `fairTradingStart`.

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 Almighty Owner Function

Note **Version 1**

The owner of the Bootstrap contract is "almighty". The function `execute` allows non-restricted calls to any contract. This setup seems fine as the owner is planned to be the `GEARBOX_TREASURY` address. The thread model assumes this address fully trusted. Still, it should be checked after deployment that all addresses were set correctly.

7.2 Fail at `FAIR_TRADING` Stage

Note **Version 1**

The function `fail` can be called by the `owner` of the contract at any stage. If it is called in `FAIR_TRADING` stage, LP tokens have already been transferred to the contract and cannot be distributed through the `claimLP` function. Instead, they have to be sent to another contract (with `execute`) on which a distribution can take place.

7.3 Inconsistent and Floating Pragma

Note **Version 1**

Gearbox Auction uses the floating pragma `^0.8.10`. Additionally, the compiler version is not set in the Foundry settings. Contracts should be deployed with consistent compiler versions and flags that were used during testing and auditing. Locking the pragma helps to ensure that contracts are not accidentally deployed using a different compiler version and helps to ensure a reproducible deployment.

7.4 Rounding Errors

Note **Version 1**

Due to rounding errors, the following can happen:

- A call to `sellGear` with 3 wei of GEAR tokens (or slightly more depending on the current shearing percentage) can be sold without the shearing fee.
- Some LP tokens might not get distributed. If, for example, 25 wei LP tokens are distributed to a user that committed all of the GEAR liquidity and all of the ETH liquidity, the user only receives 24 wei LP tokens.