PUBLIC

Code Assessment

of the Allocator Deployment Scripts

October 11, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

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

MakerDAO implements an allocation system for funding certain SubDAOs of the MakerDAO ecosystem with DAI and/or NST. This audit report reviews the security and correctness of the corresponding deployment scripts.

The most critical subjects covered in our audit are functional correctness, access control and frontrunning resistance.

Security regarding all the aforementioned subjects is high. Some Missing checks introduced small problems but these have been fixed / will be fixed as soon as it is possible.

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	1
• Code 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 following deployment scripts are part of the scope of this review:

1. deploy/AllocatorDeploy.sol
2. deploy/AllocatorInit.sol
3. deploy/AllocatorInstance.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	18 September 2023	a59ae297ef5658de432b76317621df43aefa1d09	Initial Version
2	10 October 2023	f934c91b6db00893d9a0eff420375a6ecebdf585	Second Version

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

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.

MakerDAO offers an allocation system for AllocatorDAOs of the MakerDAO system, consisting of several components for allocating NST from the MakerDAO ecosystem to each AllocatorDAO and distributing these funds in a restricted fashion. The core allocation system consists of two contracts that are governed by MakerDAO and another seven contracts that are deployed once per AllocatorDAO along with a new `ilk` in the MakerDAO core system that allows each AllocatorDAO to mint NST to their contracts. These minted NST can then be further utilized by the *Funnel* and *Conduit* subsystems.

2.2.1 MakerDAO governed contracts

`AllocatorRegistry` maps a given `ilk` to the `AllocatorBuffer` contract that belongs to the AllocatorDAO of the `ilk`.

`AllocatorRoles` enables AllocatorDAOs to delegate access to privileged functions to arbitrary addresses. While the contract is governed by the MakerDAO, each AllocatorDAO is able to set the roles for their system.

2.2.2 AllocatorDAO governed contracts

`AllocatorBuffer` holds all funds. Governance can approve other contracts for any ERC-20 token to be transferred.

`AllocatorVault` allows to mint and burn NST to and from the buffer.

Funnel system

`Swapper` allows to swap certain tokens in predefined limits in the buffer.

`DepositorUniV3` allows to supply tokens to UniSwap pools in predefined limits.

Automation system

`StableSwapper` allows to use the `Swapper` with predefined configurations.

`StableDepositorUniV3` allows to use the `depositorUniV3` contract with predefined configurations.

`ConduitMover` allows to transfer funds between *Conduits*.

Helper contracts

`AllocatorOracle` returns a price of 1M for every `ilk` that it is added to.

2.2.3 Deployment overview

The systems are deployed in two steps:

1. Some EOA deploys the contracts and - if necessary - changes the owner of these contracts to the `PauseProxy`.
2. A governance `Spell` with quorum executes the initialization of the contracts through the `PauseProxy`.

Contract deployment

The deployment is split into two parts. `AllocatorRegistry`, `AllocatorRoles` and `AllocatorOracle` are deployed once. `AllocatorVault`, `AllocatorBuffer`, `Swapper`, `DepositorUniV3`, `StableSwapper`, `StableDepositorUniV3` and `ConduitMover` are deployed once for each AllocatorDAO with references to the already deployed registry and roles contracts. During deployment, the ward of all the contracts (besides `AllocatorOracle` which is stateless) is set to Maker's *PauseProxy*.

During the initialization of the SubDAO-specific contracts, a new `ilk` is created in the MakerDAO system's `vat` and `jug` contracts``. The `AllocatorOracle` (which stays the same for all SubDAO `ilks`) is added to the `spotter` along with a liquidation factor of 1. 1M tokens of this newly generated `ilk` are then allocated to the `AllocatorVault` which is now able to `draw()` NST from the system.

`Swapper` and `DepositorUniV3` receive approvals for some predefined tokens from the `AllocatorBuffer` to be able to handle additional tokens apart from NST through the *Funnel* system.

A set of `facilitators` is further added to the `AllocatorRoles` contract giving each of the addresses the rights to `draw()` and `wipe()` on the `AllocatorVault` as well as limited use the *Funnel* contracts. Additionally, `facilitators` are set as `wards` of the automation contracts, allowing facilitators to set the configuration of the contracts and whitelist *keepers* that can operate them. `StableSwapper` and `StableDepositorUniV3` are added to the `AllocatorRoles` contract giving them rights to use the *Funnel* contracts. *Keeper* roles are added to both contracts and the `ConduitMover` contract that can then call the desired actions whenever required.

Finally, the owner of all `ilk` specific contracts is changed from the `PauseProxy` to the `AllocatorProxy` so that each AllocatorDAO can govern their own contracts.

2.2.4 Roles & Trust Model

The contracts are not deployed as proxies. However, new versions can be deployed which can be updated in the respective contracts.

New `ilks` can only be added/created by the MakerDAO governance.

The Maker *PauseProxy* is fully trusted.

The AllocatorDAOs have full control of all capital up the current `line` of their `ilk`. Their *AllocatorProxys* are therefore fully trusted.

Facilitators are partially trusted. They can execute privileged actions of limited impact. They should be limited in the damage they can do to an AllocatorDAO's balances. If they act against the SubDAO's interest, governance should have time to remove them with an emergency action before they have performed irreparable damage.

Keepers are partially/minimally trusted in that they could abuse certain configurations (e.g., extract value from slippage). They have less privilege than facilitators. If one misbehaves, they are disabled by a facilitator without the keeper being able to do critical damage.

Deployers are supposed to deploy the contracts as specified by the reviewed script. Deployers are, however, EOAs that could perform unlawful actions besides simple deployment, such as changing the settings of the system or granting themselves special privileges. It is important that after deployment, concerned parties thoroughly check the state of the deployed contracts to ensure that no unexpected action has been taken on them during deployment.

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:

- 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	1

- Missing Checks Code Corrected

Informational Findings	1

- Dynamic Debt Ceiling Missing Code Corrected

6.1 Missing Checks

Correctness Low Version 1 Code Corrected

CS-ALD-001

`AllocatorInit.initIlk()` does not check whether the `uniV3Factory` address is correctly set in the `DepositorUniV3` contract. While the deployer can not use a different, random contract instead of the correct Uniswap factory contract, they can still deploy another factory with the same bytecode. This way, `_getPool()` can still correctly determine pool addresses. The deployer is now able to set themself as the owner of the factory which in turn allows them to set and collect the protocol fee.

Also, `AllocatorInit.initIlk()` does not check whether `nstJoin` in `AllocatorVault` has been set correctly. This is documented to be added later-on but currently not implemented.

Code corrected:

The correct `uniV3Factory` address in `DepositorUniV3` is now checked in the initialization script.

Note that a check for `nstJoin` in `AllocatorVault` is still to be added when `NstJoin` has been added to the chainlog.

6.2 Dynamic Debt Ceiling Missing

Informational Version 1 Code Corrected

CS-ALD-002

The specification for the Allocator system states that:

> The Allocator Vaults have dynamic Debt Ceiling modification modules (IAMs) that are set to a medium “ttl” and medium “gap” value resulting in a throttling effect on the amount of NewStable that can be generated at once. The IAM acts as a circuit breaker in case of technical problems with the AllocatorDAOs Deployment Layer, limiting the potential for damage.

`AllocatorInit`, however, only sets a static debt ceiling and never deploys the required modules.

Code corrected:

The debt ceiling is now set to a `gap` value in the `vat`. Additionally, the `DssAutoLine` contract is setup for the new ilk by calling the `setIlk()` function with the given `gap` along with a `ttl` and a maximum debt ceiling. Calling `DssAutoLine.exec()` increases the `line` of the `ilk` by `gap` in steps of `ttl` seconds.

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 Deployment Verification

Note Version 1

Since deployment of the contracts is not performed by the governance directly, special care has to be taken that all contracts have been deployed correctly. While some variables can be checked upon initialization through the `PauseProxy`, some things have to be checked beforehand.

We therefore assume that all mappings in the deployed contracts are checked for any unwanted entries (by verifying the bytecode of the contract and then looking at the emitted events). This is especially crucial for `wards` mappings, and for `Approve` events emitted in the *AllocatorBuffer*.