PUBLIC

Code Assessment

of the Endgame Toolkit Deployment Scripts

September 27, 2024

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 Endgame Toolkit according to Scope to support you in forming an opinion on their security risks.

MakerDAO implements a toolkit for SubDAO governance including a governance token, a proxy contract for governance spell execution and a reward farming contract. 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.

In a production setting, Deployment verification is strongly recommended.

While Foundry does not atomically perform deployment, no frontrunning possibilities have been found.

The current state of the deployment and initialization scripts shows 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	3
• Code Corrected	2
• Specification Changed	1
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	3

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 Endgame Toolkit repository based on the documentation files.

The following deployment scripts are part of the scope of this review:

Version 1

```
script/CheckStakingRewardsDeploy.s.sol
script/StakingRewardsDeploy.s.sol
script/dependencies/SDAODeploy.sol
script/dependencies/StakingRewardsDeploy.sol
script/dependencies/StakingRewardsInit.sol
script/dependencies/SubProxyDeploy.sol
script/dependencies/SubProxyInit.sol
script/dependencies/VestInit.sol
script/dependencies/VestedRewardsDistributionDeploy.sol
script/dependencies/VestedRewardsDistributionInit.sol
```

Version 2

In version 2, the following files have been removed:

```
script/CheckStakingRewardsDeploy.s.sol
script/StakingRewardsDeploy.s.sol
```

The following files have been added:

```
script/01-StakingRewardsDeploy.s.sol
script/02-StakingRewardsInit.s.sol
script/09-CheckStakingRewardsDeployment.s.sol
```

Version 3

In version 3, the repository has been restructured. The following files have been removed:

```
script/01-StakingRewardsDeploy.s.sol
script/02-StakingRewardsInit.s.sol
script/09-CheckStakingRewardsDeployment.s.sol
```

The following two files have been added:

```
script/dependencies/phase-0/FarmingInit.sol
script/phase-0/01-FarmingDeploy.s.sol
```

Version 6

In version 6, the following files have been removed:

```
script/dependencies/phase-0/FarmingInit.sol
script/phase-0/01-FarmingDeploy.s.sol
```

The following files have been added:

```
script/dependencies/phase-1b/Usds01PreFarmingInit.sol
script/dependencies/phase-1b/UsdsSkyFarmingInit.sol

script/phase-1b/01-UsdsSkyFarmingDeploy.s.sol
script/phase-1b/11-Usds01PreFarmingDeploy.s.sol
```

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

V	Date	Commit Hash	Note
1	20 September 2023	e66d59a05c21bed6624e12db2b2cbda2fdb0a7e8	Initial Version
2	03 October 2023	5efa2cac3b22fa94b2599cd83cb4bfea19747091	Second Version
3	20 October 2023	5dc625fd6a07c7c24a97a45553c2287f38807e44	Updated Phase-0
4	15 November 2023	f95d2fae7992cc88ca2c6725e9ea284c895b6f3b	Refactor VestInit
5	17 January 2024	2e1d277957563400d394b03c49346aff407593c6	Refactor StakingRewards
6	28 August 2024	eb49fa619a30e4d67f46cbb21b2ef19705ff0554	Renaming and Pre-farming
7	06 September 2024	14268515aa729a588096f0d579ea38bde3e9ba2f	Minor Changes

For the solidity smart contracts, the compiler version `0.8.19` was chosen. In Version 4 the compiler version was downgraded to `0.8.16`.

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.

Version 3: Files in subfolder `phase-0-alpha` are for demo purposes only and hence out of scope of the review. Note that since Version 6, the directory has been removed.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the deployment scripts and libraries 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.

MakerDAO offers deployment scripts for the Maker Endgame toolkit, consisting of the generic SubDAO governance token *SDAO*, SubDAO governance proxy *SubProxy*, and a token farming module for NGT (and, later, SDAO) tokens.

2.2.1 SDAO deployment

An instance of the SDAO contract is supposed to be deployed for every SubDAO, representing the governance tokens of that SubDAO. For each SubDAO, the `deploy` function of library *SDAODeploy* is expected to deploy the contract and change the owner from the deployer to Maker's PauseProxy. This is, however, only planned for a later stage (Phase 1) of the Endgame plan.

2.2.2 SubProxy deployment

For each SubDAO, a SubProxy is deployed and initialized. First, the deployer creates a new SubProxy contract and changes the owner from the deployer to Maker's PauseProxy. Then the PauseProxy adds the address of the SubProxy to the chainlog. Similarly to the SDAO token, the SubProxy is planned for a later stage (Phase 1) of the Endgame plan.

2.2.3 Farming module deployment and initialization

Each farming module consists of three contracts: *DssVestMintable*, *StakingRewards*, and *VestedRewardsDistribution*. A farming module is expected to be deployed for each SubDAO. *DssVest* generates a stream of tokens to the *VestedRewardsDistribution*, which is then configured as prizes for users staking DAI/NST in *StakingRewards*.

DssVestMintable can mint an amount of NGT (SDAO in Phase 1) as its vesting stream to *VestedRewardsDistribution*. As such, it needs to be a `ward` to the NGT token to be able to call its `mint()` function. `create()` allows wards of *DssVestMintable* to create streams of tokens towards any address. As such, the only ward after deployment has to be the trusted Maker's PauseProxy. `cap`, the maximum amount of tokens per second that are streamed in a vest, has to be increased from the default 0.

StakingRewards is constructed with an `owner` address that can pause the contract and is expected to be Maker's PauseProxy, the `rewardsDistribution` which is an address that can notify new rewards and is to be *VestedRewardsDistribution*, `rewardsToken` which is the address of the NGT token, and `stakingToken` which is the token staked for farming (either DAI or NST).

VestedRewardsDistribution receives a vesting stream of tokens from *DssVest* and transfers them as rewards to *StakingRewards*. Its only deployment parameters are therefore the addresses of these two contracts.

Since *StakingRewards* and *VestedRewardsDistribution* depend on each other's addresses for correct configuration, *StakingRewards* is first deployed, setting its `rewardsDistribution` field to the zero address. Then *VestedRewardsDistribution* is deployed correctly referencing *StakingRewards*, and finally `rewardsDistribution` in *StakingRewards* is set to the address of *VestedRewardsDistribution*.

After the contracts are deployed, a vesting stream can be created by the DssVest owner with *DssVest* as beneficiary, no cliff period, and the *restricted* field set to one.

Note: The aforementioned contracts are deployed and initialized by an EOA and the PauseProxy is not set as `ward` to conduct end-to-end tests in a first step. The examined deployment entrypoint scripts will be used as templates for Spells that will be used to integrate the contracts into the Maker ecosystem in Phase 0.

2.2.4 Trust model & Roles

In the current state (as of this writing), the farming contracts will be owned by an EOA (or, in case of DAO and SubProxy, not be deployed at all). The contracts should therefore be considered completely trusted.

After deployment in Phase 0/1, every contract's ownership is assumed to be transferred to Maker's *PauseProxy*, and no other wards are maintained. No privileged actions happen between deployment and transfer of ownership, such as minting of tokens, creation of vests, notification of rewards, etc. 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.

For SDAO contract deployed on L2, it is expected that the L2 Governance Relay and the L2 Token Bridge are the only wards.

2.2.5 Changes in Version 3

Actual deployment scripts for the farming module have been added: 01-Farming-Deploy.s.sol and FarmingInit.sol. Their code is based on previously existing scripts for test / demo deployment.

The new 01-Farming-Deploy.s.sol script executed off-chain deploys a StakingRewards and VestedRewardsDistributon contract if no respective address is already present in the local `FOUNDRY_CHANGELOG.changelog` file.

For the StakingRewardsContract, the owner is set to the `MCD_PAUSE_PROXY`, the staking token is the `NST` and the rewards token is the `NGT` token.

For the initialization, library `FarmingInit` is provided. This code is intended to be executed on chain in the execution context of the `MCD_PAUSE_PROXY`, hence it has the necessary privileges to execute the actions.

Given the inputs:

```
address nst;
address ngt;
address rewards;
address dist;
address vest;
uint256 vestTot;
uint256 vestBgn;
uint256 vestTau;
```

sanity checks are performed to ensure the given contracts configuration is compatible. Additionally it is ensured that the `vest`, the vesting contract assumed to be an instance of a DSS-Mintable-Vest, has minting rights on the NGT token.

The StakingRewards contract is initialized using the functionality of the StakingRewardsInit library. The VestedRewardsDistributor contract is set as the distributor.

A new vesting stream in `vest` is created given the input parameters, afterwards the VestedRewardsDistributor updated accordingly, it's local parameter `vestId` is set to the id of the created vesting stream.

2.2.6 Changes in Version 5

In version 5 an additional check has been added to the init script `FarmingInit` to ensure the reward token isn't equal to the staking token.

2.2.7 Changes in Version 6

In version 6, NST and NGT has been renamed to USDS and SKY respectively in the scripts. The deployment scripts now retrieve the PauseProxy address from the chainlog, and the init scripts now add the reward and distribution contracts to the chainlog.

In addition, pre-farming deploy and init scripts are added to deploy a `StakingRewards` contract with USDS as staking token, `address(0)` as rewards token, and Maker's PauseProxy as the owner. The

pre-farming contract is used to keep the history and all the events on-chain, and the rewards calculation and distribution will be achieved off-chain.

2.2.8 Changes in Version 7

`VestInit.init()` has been removed. Setting the `cap` is expected to be done manually by calling `file()`. Additionally, the sanity checks in the scripts have been adjusted:

1. `UsdsSkyFarmingInit`: The inequality of `rewardsToken` and `stakingToken` is not validated anymore as this is part of the constructor. Further, the comparison against the parameters passed by governance (expected to be distinct) further ensures their inequality since governance is expected to provide correct parameters.
2. `UsdsSkyFarmingInit`: The reward contract's owner is validated. Note that this check is now explicit and has previously not been directly visible (as part of the call to `setRewardsDistribution()`).
3. `UsdsSkyFarmingInit` and `Usds01PreFarmingInit`: It is not validated that the last update time is zero. However, it is now validated that the reward rate is zero which results in a similar property (since a zero update time would have implied a reward rate of zero). Additionally, the reward distribution is ensured to be `0x0`. For the former, in case of no unexpected updates, the reward distribution would have remained without effect as it would have been overwritten afterwards. For the latter, the post-initialization storage will be cleaner. Additionally, the changes aim to prevent a permissionless DoS vector. Namely, `getReward()` could have allowed for permissionless updates to the last update time if the period finish storage value had been set (which, however, requires permissions). Ultimately, the validation process for governance might be simplified.

In the latest version, the same SDAO contract for Spark will also be deployed on L2.

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:

- 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	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	3

- StakingRewards rewardsDistribution Ownership Inconsistency Specification Changed
- Vest Minting Not Possible Code Corrected
- Vest Ownership Not Transferred at Deployment Code Corrected

Medium-Severity Findings	0
Low-Severity Findings	3

- Missing Checks Code Corrected
- SubProxy rely() to MCD End Instead of MCD ESM During Initialization Code Corrected
- Vest Should Not Have a Cliff Period Code Corrected

Informational Findings	1

- Redundant Imports Code Corrected

6.1 StakingRewards `rewardsDistribution` Ownership Inconsistency

Correctness High Version 1 Specification Changed

CS-EGTKD-001

`StakingRewardsInit.init()` is called by the deployer after `StakingRewardsDeploy.deploy()` has been called, setting its owner to `p.owner`, which should be Maker's PauseProxy. `StakingRewardsInit.init()` is therefore not called by the owner (as the Foundry script cannot be run by a governance Spell) and will revert. If `p.owner` is the deployer, then the ownership is not correctly transferred to the PauseProxy anywhere in the script.

Specification changed:

MakerDAO informed us that the audited deployment script is currently meant for testing purposes. The deployer will therefore be an EOA. This will change later in Phase 0 of the Endgame Plan, where the contracts are initialized via governance Spell setting the owner of the contracts to the PauseProxy. The deployment scripts will be used as templates for the final Spell.

6.2 Vest Minting Not Possible

Correctness High Version 1 Code Corrected

CS-EGTKD-002

`DssVestMintable.pay()` calls the `NGT` token's `mint()` function to generate tokens for the vesting. The functions is guarded and can only be accessed by a `ward`. The `DssVestMintable` contract is never set as a ward of the `NGT` contract.

Code corrected:

The initialization script now performs the following call, setting the ward of the `NGT` contract:

```
RelyLike(ngt).rely(vest);
```

This call is only possible if the deployer is an EOA that has been set as `ward` in the `NGT` contract. Since the script is currently only deploying contracts for testing purposes, the supplied `NGT` contract will have the correct rights in the given environment.

6.3 Vest Ownership Not Transferred at Deployment

Security | High | Version 1 | Code Corrected

CS-EGTKD-003

StakingRewardsDeploy deploys *DssVestMintable*, whose ward has unlimited token minting ability for the vested token by creating arbitrary new vests. The `ward` of *DssVestMintable* is not transferred to Maker's PauseProxy after deployment. It remains at the address of the deployer.

Code corrected:

The owner of *DssVestMintable* is now transferred to the given `admin` address of the deployment script. Since the deployment script is, in a first step, run by an EOA and only used for testing purposes, the ownership will not be transferred to the PauseProxy. This will be different later in Phase 0 of the Endgame plan.

6.4 Missing Checks

Security | Low | Version 2 | Code Corrected

CS-EGTKD-007

`Phase0StakingRewardsInitScript` does not check the correct state of some of the deployed contracts. In particular, the following checks are missing:

- `stakingToken` in `StakingRewards` is not checked to be the actual NST contract.
- `dssVest` and `stakingRewards` in `VestedRewardsDistribution` are not checked to be equal to the actual `DssVestMintable` and `StakingRewards` contracts.
- It is not checked that the `rewardRate` in `StakingRewards` has already been updated (e.g., by checking that `lastUpdateTime` is 0). This is possible if the deployer adds their own rewards distribution contract and calls `notifyRewardAmount` with it.

Code corrected:

While originally the scripts related to the deployment and initialization of the farming module (including `Phase0StakingRewardsInitScript.sol` of the issue above) have been intended for testing/demo purposes only, in Version 3 these scripts have been adapted to be used for the actual deployment in phase 0. The missing checks have been added to the code.

6.5 SubProxy `rely()` to MCD End Instead of MCD ESM During Initialization

Correctness Low Version 1 Code Corrected

CS-EGTKD-006

In the *SubProxyInit* library, the `init()` function sets MCD End as a ward of the SubProxy. MCD End has no ability to administrate arbitrary contracts, such as the SubProxy in question. The purpose of the `rely()` is therefore unclear.

Code corrected:

MCD End has been replaced with MCD ESM (Emergency Shutdown Module) which will be able to remove the PauseProxy from the `wards` of the contract.

6.6 Vest Should Not Have a Cliff Period

Design Low Version 1 Code Corrected

CS-EGTKD-004

VestedRewardsDistribution requires the configured vest to not have a cliff period past the beginning. *StakingRewardsDeploy* however supports a non-zero value for `vestEta`.

Code corrected:

The `vestEta` option has been removed.

6.7 Redundant Imports

Informational Version 3 Code Corrected

CS-EGTKD-005

`VestInit.sol` imports `dss-test/ScriptTools.sol` that is redundant. It is never used in this library, in addition, it is built on top of the forge standard library that can only be used for off-chain testing.

Code corrected:

The redundant import has been removed.

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

Note: This is only relevant for the deployment in Phase 0/1 of the Endgame plan.

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.

In the case of `DssVestMintable`, special care also has to be taken to make sure that no extra `awards` have been added by the deployer. During initialization, the PauseProxy adds the contract as a `ward` to the NGT contract. After this, if the deployer added any `awards` with a controlled address as `usr`, they are able to mint tokens to themselves.