PUBLIC

Code Assessment

of the Endgame Toolkit Smart Contracts

January 16, 2026

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 14
4 Terminology 15
5 Open Findings 16
6 Resolved Findings 18
7 Informational 22
8 Notes 23

1 Executive Summary

Dear all,

Thank you for trusting us to help Sky 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.

Sky 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 contracts as well as the corresponding deployment scripts.

In the latest version reviewed changes were made to update a farm's existing vest. Overall the endgame-toolkit offers a new governance token for SubDAO-level governance, a SubProxy for executing governance delegatecalls and a farming module allowing stakers to earn rewards.

The most critical subjects covered in our audit are functional correctness, access control and frontrunning resistance. While security regarding all the aforementioned subjects is high, this reports contains some notes about the proper use of the contracts.

In a production setting, Deployment verification is strongly recommended.

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

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	3
• Code Corrected	2
• Specification Changed	1
Medium-Severity Findings	0
Low-Severity Findings	4
• Code Corrected	3
• Risk Accepted	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 Endgame Toolkit repository based on the documentation files.

The scope for Version 1 consists of the two solidity smart contracts:

```
./src/SDAO.sol
./src/SubProxy.sol
```

In Version 2 the scope was extended to include the farming module:

```
./src/VestedRewardsDistribution.sol
./src/synthetix/StakingRewards.sol
```

For the StakingReward contract, the focus was on validating that the upgraded contract is equivalent to the original one. This is not a complete review of the Synthetix StakingReward contract.

In Version 4 the following deployment scripts are part of the scope of this review:

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

In Version 5, the following files have been removed:

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

The following files have been added:

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

In Version 7, the repository has been restructured. The following files have been removed:

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

The following two files have been added:

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

In **Version 10**, the following files have been removed:

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

The following files have been added:

```
script/
    dependencies/
        phase-1b/
            Usds01PreFarmingInit.sol
            UsdsSkyFarmingInit.sol
    phase-1b/
        01-UsdsSkyFarmingDeploy.s.sol
        11-Usds01PreFarmingDeploy.s.sol
```

In **Version 12**, the following files have been added:

```
script/
    dependencies/
        phase-1d/
            LsmkrSpkFarmingInit.sol
            SkySpkFarmingInit.sol
            SpkSkyFarmingInit.sol
            UsdsSpkFarmingInit.sol
    phase-1d/
        01-UsdsSpkFarmingDeploy.s.sol
        11-SkySpkFarmingDeploy.s.sol
        21-SpkSkyFarmingDeploy.s.sol
        31-LsmkrSpkFarmingDeploy.s.sol
```

In **Version 13**, the deployment script for SPK has been added:

```
script/
    phase-1d/
        40-SpkDeploy.s.sol
```

In Version 14, the following script has been added:

```
script/dependencies/treasury-funded-farms/TreasuryFundedFarmingInit.sol
```

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

V	Date	Commit Hash	Note
1	02 May 2023	da937582c5b8ca444fd31627f91a6fa5ede35d92	Initial Version
2	28 Jun 2023	ab305de703e51a3523b18991cd136f4c1fc1298b	Farming Module
3	10 Jul 2023	f0919fd1e6e1ea933fd42eaf24840fb40da797df	Fixed Typo
4	20 Sep 2023	e66d59a05c21bed6624e12db2b2cbda2fdb0a7e8	Initial Deployment Scripts
5	03 Oct 2023	5efa2cac3b22fa94b2599cd83cb4bfea19747091	Updated Deployment Scripts
6	04 Oct 2023	cc223aa66ca7a5da38219beab6363f6b3eb44efb	Updated SDAO
7	20 Oct 2023	5dc625fd6a07c7c24a97a45553c2287f38807e44	Updated Phase-0
8	15 Nov 2023	f95d2fae7992cc88ca2c6725e9ea284c895b6f3b	Refactor VestInit
9	17 Jan 2024	2e1d277957563400d394b03c49346aff407593c6	Refactor StakingRewards
10	28 Aug 2024	eb49fa619a30e4d67f46cbb21b2ef19705ff0554	Renaming and Pre-farming
11	06 Sep 2024	14268515aa729a588096f0d579ea38bde3e9ba2f	Minor Changes
12	09 Oct 2024	5bf4b1771b99f5f8758fd40a4ac567f797b5405b	SPK Farming
13	09 Dec 2024	e6c3a783614748717b4cb8d671c907a1feb71121	SPK Deployment
14	13 Jan 2026	4f238f9b23298190150d49482bad56c00f0af825	Update Farm Vest

For the solidity smart contracts, the compiler version `0.8.19` was chosen. In Version 8 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 7: Files in subfolder `phase-0-alpha` are for demo purposes only and hence out of scope of the review. Note that since Version 10, the directory has been removed.

2.2 System Overview

This system overview describes the initially received contracts (Version 1) and deployment scripts (Version 4) as defined in the Assessment Overview.

At the end of this report section we have added a changelog 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.

Sky implements a toolkit for SubDAO-level governance with:

- A mintable ERC-20 SDAO token for SubDAO governance, which supports EOA and smart contract signature validation for approvals.
- A SubProxy for executing governance delegatecalls, which isolates the context of execution for spells from the main governance contract to avoid potential exploits messing with the original contract storage.
- A farming module allowing to claim rewards when staking tokens.

Sky also offers deployment scripts for these components.

2.2.1 SDAO

`SDAO` is an ERC-20 token for governance with 18 decimals. The contract is controlled by privileged roles `wards`, which is initialized with `msg.sender` in the constructor. Any address in `wards` has owner access to:

- Add a new ward by `rely()`.
- Remove a ward by `deny()`.
- Mint any amount of `SDAO` tokens to an address by `mint()`.

Token transfers work the same way as a normal ERC-20 token but with a few restrictions. Specifically, transfers to the zero address (address(0)) or the contract itself are not allowed. A user can also burn its own tokens by calling `burn()` with its own address. In case the address specified is different to the `msg.sender`, the user will burn on behalf of others if its allowance is sufficient.

SDAO supports the unlimited allowance pattern. In addition, `permit()` is provided for setting allowance with signatures either from an EOA or a contract (EIP-1271). A contract can give permission to a spender by implementing `isValidSignature()` with customized verification logic. If the signature length does not equal to 65 bytes, it is assumed the allowance owner is a contract, which will be queried for signature validation.

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

`SubProxy` is the SubDAO-level *PauseProxy*. This proxy uses *delegatecall* to execute calls from context isolated from the main governance contract. All the contracts controlled by the SubDAO must authorize this proxy instead of the governance contract itself. The proxy itself is controlled by the wards initialized with `msg.sender` in the constructor. Different from Maker's DSPauseProxy where there is only one owner (the DSPause contract), any address in `SubProxy`'s `wards` has owner access to:

- Add a new ward by `rely()`.

- Remove a ward by `deny()`.
- Trigger a delegatecall execution on the target address by `exec()`.

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

Users will be able to farm reward tokens by staking their assets. This implementation reuses existing code: DssVest to permissionlessly distribute the allocated funds according to certain rules and StakingRewards forked from Synthetix to implement the on-chain reward system. These contracts are connected through another contract called VestedRewardsDistributon.

`StakingRewards`:

Implements the on-chain reward system for stakers. The implementation is a fork of the well known and battle tested Synthetix implementation. Major changes include:

- `stake(uint256 amount, uint16 referral)` which additionally emits an event emitting the referral code. Referral rewards are handled off-chain.
- Update to Solidity 0.8.x, the code has been refactored accordingly.

The contract implements the following functionality:

- `stake()`: Allows users to stake. Stake is represented by an non-transferable ERC20 like token interface.
- `withdraw()`: Allows users to withdraw their stake.
- `getReward()`: Allows users to claim their rewards.
- `exit()`: Allows users to withdraw their stake and claim their rewards.

Stakers earn rewards for staking during reward distribution periods only. The mechanism works as follows: Reward tokens pushed to the contract are released to the stakers linearly and proportional to the balance staked over a period called reward duration. All actions changing the staked balance of an account must update the earned reward for the account to ensure the accounting is correct.

- `notifyRewardAmount()`: Permissioned function called by the RewardsDistributor after having transferred the new batch of reward tokens to be distribution over the next period. This function supports to be called both either outside of a rewards distribution period or within an ongoing period. A new reward distribution is started taking all funds to be distributed into account.

Through `setRewardsDuration()` and `setRewardsDistribution()` the owner can update the parameters governing the reward distribution.

`VestedRewardsDistributor`:

Connects the DssVest contract releasing the funds for the rewards according to a vesting schedule and the StakingRewards contract. This contract must be set as beneficiary of the `vestId`.

- `distribute()`: Permissionless function, if the conditions are fulfilled (proper `vestId` set, unpaid funds available) claims the outstanding rewards, forwards them to and notifies the StakingRewards contract.

The privileged role (assumed to be the Governance SubProxy) can update the `vestId`. This must be done after setup to activate the contract. If a vestId of the DssVest expires, the vestId can simply be updated.

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 Roles and Trust Model

The wards of SDA0 token are trusted to not misbehave, otherwise any amount of tokens can be minted at its discretion. The `SubProxy` contract would be authorized by contracts under SubDAO's control for privileged operations, hence the wards of the proxy are assumed to behave honestly and correctly at all times and never act against the interest of the system users. (e.g. selfdestruct the proxy, abuse authorization, etc.).

The `owner` of StakingRewards and the wards of the VestedRewardsDistribution are trusted to not misbehave.

Users (Token holder / stakers): Untrusted

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 Changelog

2.2.5.1 Changes in Version 2

- The token parameters `name` and `symbol` can now be updated by the wards.

2.2.5.2 Changes in Version 6

- Functions `increaseAllowance` and `decreaseAllowance` have been removed. Only functions `approve` and `permit` remain to modify allowances.
- Permit functionality: Now, when validating a signature with a contract, if there is no code deployed at the given address, the transaction will revert with a clear error message. Previously the transaction would just revert.

2.2.5.3 Changes in Version 7

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.

2.2.5.4 Changes in Version 9

`setRewardsDuration()` now allows to change the rewards duration during an active distribution. `rewardRate` and `periodFinish` are recalculated and updated accordingly.

The constructor of StakingRewards now enforces the reward and staking tokens to be distinct. Issuing rewards in the same tokens as the staking token is not supported and would compromise the contract's accounting; this added check effectively safeguards against improper configuration.

Additionally, it was defined that SDAO tokens are additionally intended to be used on L2s.

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

2.2.5.5 Changes in Version 10

In this version, 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.

2.2.5.6 Changes in Version 11

`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.

2.2.5.7 Changes in Version 12

Deployment scripts and initialization libraries have been added for the following SPK (Spark) related farms:

1. SkySpkFarming: stake SKY to farm SPK as reward.
2. SpkSkyFarming: stake SPK to farm SKY as reward.
3. UsdsSpkFarming: stake USDS to farm SPK as reward.
4. LsmkrSpkFarming: stake lsMKR to farm SPK as reward. Note that this farm is to be used by LockstakeUrn contracts.

These farming contracts will be deployed and initialized with the same approach as UsdsSkyFarming.

2.2.5.8 Changes in Version 13

A deployment script for the SPK token has been added.

2.2.5.9 Changes in Version 14

A deployment script for transferable vest initialization and existing farm's vest update has been added.

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 Open 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

- Precision Loss in rewardRate Calculation Risk Accepted

5.1 Precision Loss in rewardRate Calculation

Design Low Version 1 Risk Accepted

CS-MET-001

The calculation of the `rewardRate` causes a potentially harmful loss of precision. The default `rewardsDuration` denominator (1 week reward duration) loses up to `604800` wei of precision every time `notifyRewardAmount()` is called, in the following calculation:

```
rewardRate = reward / rewardsDuration;
```

and

```
rewardRate = (reward + leftover) / rewardsDuration;
```

The amount lost due to rounding has been deposited in the contract, but the internal accounting loses track of it, rendering it unclaimable.

If `rewardsToken` is a token with a high value per wei, the loss can be significant. For example, if `rewardsToken` is `USDC`, which has 6 decimals, the loss can be of up to `$ 0.6048` every time `notifyRewardAmount()` is called. If the token is `WBTC`, which has 8 decimals but much higher value per wei, the loss is up to `$ 181` every time the `rewardRate` is computed. Since `notifyRewardAmount()` can be called every 12 seconds through *VestedRewardsDistribution*, the rounding amount can be lost up to 50400 times per week.

For tokens with a higher number of decimals and a low value per wei, for example `DAI` or `WETH`, the loss is less significant. It amounts to a maximum of `$ 0.00006` per week for `WETH` and `$ 3e-18` per week for `DAI`. The maximum weekly loss can be calculated as follows:

```
weeklyLoss = rewardsDuration * tokenValuePerWei * blocksPerWeek
```

In Version 9 `setRewardsDuration()` was updated to allow updating the `rewardRate` during an active distribution. The calculation `rewardRate = leftover / _rewardsDuration;` is subject to similar rounding effects / loss of precision. Notably the parameter `_rewardsDuration` is controlled by the caller of the function, which is restricted to the trusted owner. The caller must ensure the resulting loss in precision is acceptable.

Risk accepted:

Sky states:

```
Risk accepted. We are aware of the issue with precision loss, however we
wanted to avoid making changes to the original code as much as possible. The
StakingRewards contract in this context will only ever handle tokens with 18
decimals (DAI, MKR, SubDAO tokens, NewStable – Dai equivalent, NewGov – MKR
equivalent). If we take MKR as an example, its all-time high price was just
short of 6,300 USD. Let’s extrapolate its value imagining it could grow 100x
for the duration of the staking rewards program. Using the formula you provided,
we would have:

weeklyLoss  = rewardsDuration * tokenValuePerWei * blocksPerWeek
            = 604800 * 50400 * (630000 * 10^(−18))
            = 0.0192036096

Even in this extreme scenario, weekly losses would amount to less than 0.02 USD,
which is acceptable for us.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open 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	2

- Redundant Imports Code Corrected
- Typo in Documentation Code Corrected

6.1 StakingRewards `rewardsDistribution` Ownership Inconsistency

Correctness High Version 4 Specification Changed

CS-MET-005

`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:

Sky 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 4 | Code Corrected

CS-MET-003

`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 4 | Code Corrected

CS-MET-004

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 5 | Code Corrected

CS-MET-009

`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 7 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 4 Code Corrected

CS-MET-008

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 4 Code Corrected

CS-MET-006

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 7 Code Corrected

CS-MET-007

`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.

6.8 Typo in Documentation

Informational | Version 1 | Code Corrected

CS-MET-002

In the NatSpec for the *VestedRewardsDistribution* contract at line 23 `RewardsDistribution` is misspelled.

Code corrected:

The typo has been corrected.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Approve May Fail on Weird Tokens

Informational **Version 14** **Acknowledged**

CS-MET-010

In TreasuryFundedFarmingInit, `rewardsToken.approve()` will be called to increase the allowance for vesting. Note:

1. The `approve()` return value is not checked, consequently it may fail silently.
2. The token may require an approve call with 0 before changing a non-zero allowance (e.g. USDT), hence `approve()` may revert.

Hence only legitimate tokens are expected to be used without these weird behaviors.

Acknowledged:

Sky is aware and expects only standard tokens.

8 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.

8.1 Ability to Modify `name` & `symbol`

Note Version 2

In this version, functionality for the privileged role to update the token parameter `name` and `symbol` is introduced. Note that is unusual for ERC20 tokens and must be done with care. Some downstream applications or smart contracts may not be designed to accommodate such changes.

Consider these illustrative examples:

- Upon deployment the name of Curve pools is set using the traded token names.
- The representative token deployed by third-party bridges to other chains is often based on the original token's name and symbol.

8.2 Deployment Verification

Note Version 4

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.

8.3 Rewards in StakingRewards Might Take Longer to Vest Than Expected

Note Version 1

Rewards added to StakingRewards could be expected to be payed out to stakers in `rewardsDuration` time, which is initially set to 7 days. However, every time `notifyRewardAmount()` is called, a new `rewardRate` is computed prolonging the vesting of the remaining amount over the next `rewardDuration`.

As a simple example, assume we are distributing 1000 DAI over one week, then the `rewardRate` will be `rewardRate = 1000 * 10**18 / rewardDuration`. If after 3.5 days have passed we call `notifyRewardAmount()`, adding a `0` reward, the new `rewardRate` is computed as

```
rewardRate = (reward + leftover) / rewardsDuration;
```

which will amount to `rewardRate = 500 * 10**18 / rewardDuration`. Moreover, the periodFinish will be pushed back by `rewardDuration`, moving it from `initialTime + rewardDuration` to `initialTime + rewardDuration/2 + rewardDuration`. Overall, the reward distribution will last 1.5 times the expected duration, with the latter `rewardDuration` period having half the effective rewardRate as expected.

If we take this reasoning to the extreme, `notifyRewardAmount()` can be called every block, which will every time increase the `periodFinish` by 12 seconds, and reduce the reward rate by `1 - blockDuration/rewardDuration`. This is because the new rewardRate will be the old reward minus the consumed reward.

$$rewardRate_t = \frac{(reward_{t-1} - rewardRate_{t-1} * blockDuration)}{rewardDuration}$$

$$= \frac{reward_{t-1} - \frac{reward_{t-1}}{RewardDuration} * blockDuration}{rewardDuration}$$

$$= \frac{reward_{t-1}}{rewardDuration} * (1 - \frac{blockDuration}{rewardDuration})$$

$$= rewardRate_{t-1} * (1 - \frac{blockDuration}{rewardDuration})$$

Over n block the rewardRate will decrease from the initial rewardRate by `(1-blockDuration/rewardDuration)^n`, which is an exponential decay for the rewardRate, corresponding to an exponential decay of the remaining reward. The reward will therefore not be distributed in a finite amount of time. Numerical simulations have showed that after 1 week 63% will have been distributed, after 2 weeks 86%, after 3 weeks 95%, after 4 weeks almost 99%.

In practice, anybody can trigger `notifyRewardAmount()` at every block by calling the `distribute` method of *VestedRewardsDistribution*, the cost of doing so in terms of gas is likely to offset any advantage that such an attacker can get from delaying in such a way the reward rate.

Calling `distribute()` every block will however not pass an amount of zero `notifyRewardAmount()`, but it will pass the reward per block vested in *dssVest*. In the steady state, when the dssVest has been supplying a constant stream of reward for a long time, even factoring in the exponential decay behavior, the `rewardRate` in *StakingRewards* will converge to the same constant rate as in *dssVest*.