PUBLIC

Code Assessment

of the EndGame Toolkit Smart Contracts

July 10, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Maker 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.

In the latest version reviewed Maker added a farming module allowing stakers to earn rewards. Furthermore this endgame-toolkit offers a new governance token for SubDAO-level governance and a SubProxy for executing governance delegatecalls.

The most critical subjects covered in our audit are security, functional correctness and seamless integration with the existing system. While security regarding all the aforementioned subjects is high, this reports contains some notes about the proper use of the contracts. The most significant finding discusses Precision Loss in rewardrate Calculation.

In summary, we find that the codebase provides a good 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
• 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:

1. ./src/SDAO.sol
2. ./src/SubProxy.sol

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

1. ./src/VestedRewardsDistribution.sol
2. ./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.

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

V	Date	Commit Hash	Note
1	2 May 2023	da937582c5b8ca444fd31627f91a6fa5ede35d92	Initial Version
2	28 June 2023	ab305de703e51a3523b18991cd136f4c1fc1298b	Farming Module
3	10 July 2023	f0919fd1e6e1ea933fd42eaf24840fb40da797df	Fixed Typo

For the solidity smart contracts, the compiler version `0.8.19` 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

Maker 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.

2.2.1 SDAO

SDAO is an ERC-20 token for governance with 18 decimals. The contract is controlled by priviledged 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.

Changes in Version 2:

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

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()`.

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 referal 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-transferrable 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 fullfilled (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.

2.2.4 Roles and Trust Model

The wards of SDAO 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 priviledged 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

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 our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies

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
```

Risk accepted:

Maker 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 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.

6.1 Typo in Documentation

Informational **Version 1**

CS-MET-002

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

The typo has been corrected.

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 Ability to Modify `name` & `symbol`

Note **Version 1**

Version 2 introduces functionality for the priviledge role to update the token parameter `name` and `symbol`. 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.

7.2 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.

$$
\begin{aligned}
rewardRate_t &= \frac{(reward_{t-1} - rewardRate_{t-1} * blockDuration)}{rewardDuration} \\
&= \frac{reward_{t-1} - \frac{reward_{t-1}}{RewardDuration} * blockDuration}{rewardDuration} \\
&= \frac{reward_{t-1}}{rewardDuration} * (1 - \frac{blockDuration}{rewardDuration}) \\
&= rewardRate_{t-1} * (1 - \frac{blockDuration}{rewardDuration})
\end{aligned}
$$

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*.

7.3 Vesting Plan Must Be Restricted

Note **Version 1**

If a vesting plan of DSSVest is restricted, that means only the recipient of the rewards may claim them, no one else can trigger the distribution of the rewards.

For the correct operation of VestedRewardsDistribution it's important that the plan is restricted: `VestedRewardsDistribution.distribute()` forwards the amount retrieved in `amount = dssVest.unpaid(vestId);` only, any excess balance held at the contract is not forwarded.