CANTINA

Maker Dao

Security Review

Cantina Managed review by:

Christoph Michel, Lead Security Researcher

M4rio.eth, Security Researcher

November 19, 2023

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Maker Protocol, also known as the Multi-Collateral Dai (MCD) system, allows users to generate Dai (a decentralized, unbiased, collateral-backed cryptocurrency soft-pegged to the US Dollar) by leveraging collateral assets approved by the Maker Governance, which is the community organized and operated process of managing the various aspects of the Maker Protocol.

From Sep 22nd to Sep 25th the Cantina team conducted a review of endgametoolkit on commit hash e66d59a0. The team identified a total of **13** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 5
- Gas Optimizations: 0
- Informational: 7

3 Findings

3.1 Medium Risk

3.1.1 `StakingContract.setRewardsDuration` can be DoS'd

Severity: Medium Risk

Context: StakingRewards.sol#L165

Description: The `StakingRewards.setRewardsDuration` function can only be called after `periodFinish`. However, anyone can call `VestedRewardsDistribution.distribute` which calls `StakingReward.notifyRewardAmount` and sets a new `periodFinish`. It's unlikely that the `rewardDuration` can ever be changed as long as the `DssVest` is still active.

Recommendation: Be aware that you're unlikely to be able to change the `rewardsDuration` once the `DssVest` vesting starts. Alternatively, allow changing the rewards duration even before the current period finishes by adding reward rebasing logic to the `setRewardsDuration` by:

1. Accumulating the current rewards via `updateReward(address(0))`.
2. Calculating the old reward `leftover`.
3. Changing the `rewardsDuration`.
4. Setting the new `rewardRate` over the new duration, and setting the new `periodFinish` and `lastUpdateTime`.

There is currently a workaround by first calling `setRewardsDistribution(address(0))` to disable `notifyRewardAmount` calls, and waiting until the period is over. Then `setRewardsDuration` can be called, and the rewards distribution contract can be set again. However, this leads to non-smooth reward distributions.

Maker: Acknowledged. We wanted to avoid changes to `StakingRewards` as much as possible. While we did not anticipate the problem you highlighted, we do not intend to modify the rewards duration. If there is ever a need to do it, the workaround would work fine.

Cantina: Acknowledged.

3.2 Low Risk

3.2.1 Arbitrary `delegatecall` within the `SubProxy`

Severity: Low Risk

Context: SubProxy.sol#L76

Description: Within the `SubProxy` the `delegatecall` is used to call various functions on the contracts that the `SubProxy` is the owner. This `delegatecall` is very permissive as it does not have a target or args, both of them being received as parameters. This can be very dangerous if a malicious call is passed through as there are no check on the actual target or arguments. We assume this function will be under governance and it will require a governance proposal in order to be called which makes the issue a low severity as the governance proposal can be audited before being called.

Recommendation: Consider handling this with precautions, consider doing mainnet forking tests as well before running any call and analyze the results.

Maker: Acknowledged. This permissive behavior is required/desired because this is a Governance-controlled proxy used to isolate the context of execution for SubDAO spells. In that sense, `SubProxy` is no different from `DsPauseProxy`, which have been used in Maker Protocol pretty much since its inception. We are aware of the security implications, and we do have a proper process to handle the execution through it.

Cantina: Acknowledged.

3.2.2 The rewards will be lost between first `notifyRewardAmount` and first stake

Severity: Low Risk

Context: StakingRewards.sol#L102

Description: When the `notifyRewardAmount` is called, we assume that the amount of reward starts to be distributed at the `block.timestamp`. This is not true if the `_totalSupply` is 0, meaning no stakes are present in the contract. For the first staker, the reward period starts at the moment it called `stake`. Until that point, `rewardPerToken` is 0 because `_totalSupply` is 0.

```
function rewardPerToken() public view returns (uint256) {
  if (_totalSupply == 0) {
      return rewardPerTokenStored;
  }
  return
      rewardPerTokenStored + (((lastTimeRewardApplicable() - lastUpdateTime) * rewardRate * 1e18) /
↪  _totalSupply);
}
```

This means that between the first reward distribution and the first stake, the reward will be lost as there is no stacker to claim that period.

Recommendation: The rewards can be retrieved via `recoverERC20`. Another way of mitigating this scenario without modifying the code is to start the `DSSVest` with a `bgn` in the future so the users have time to stake before the rewards start being distributed. Another recommendation which requires the code to be modified, is to define the `periodFinish` when the first stake is happening, which means the `_totalSupply` is zero.

Maker: Acknowledged. We prefer not to make further modifications to the code and handle this from an operational perspective. `bgn` will be set to a date in the future and worst case scenario, any team member involved in the launch could stake a minimum amount of tokens to prevent this issue from happening.

Cantina: Acknowledged.

3.2.3 Solidity rounding dust can accumulate in the `StakingRewards`

Severity: Low Risk

Context: StakingRewards.sol#L139

Description: The pattern used in the Synthetix `StakingRewards` contract is push pattern. The `amount` of the rewards are pushed to the contract and then the `notifyRewardAmount` is called with the `amount`.

The rewards then are distributed via a `rewardRate` which is updated accordingly with the `rewardsDuration`.

```
if (block.timestamp >= periodFinish) {
    rewardRate = reward / rewardsDuration;
} else {
    uint256 remaining = periodFinish - block.timestamp;
    uint256 leftover = remaining * rewardRate;
    rewardRate = (reward + leftover) / rewardsDuration;
}
```

Due to the rounding down of solidity, the `rewardRate * rewardsDuration` should be == `reward` but it will be slightly less, depending on how big the `rewardsDuration` is.

Recommendation: A way to combat this is to consider the dust accumulated on the previous epoch of distribution as added amount to the `reward` on the new epoch. If the aforementioned recommendation is not desired as it will complicate the `notifyRewardAmount`, another way will be to recover the dust accumulated over a longer period of time using the `recoverERC20`. With this approach it might be complicated on how much dust is accumulated over the time if new epochs are coming in or not all the rewards were claimed.

Maker: Acknowledged We are aware of this issue, however this contract is only meant to be used with contract having 18 decimals, making the rounding issues negligible. The same issue was pointed out by ChainSecurity in their report on Section 5.1.

Here is the relevant section of our response to them:

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 to us.

Cantina: Acknowledged.

3.2.4 Reward tokens can be retrieved from `StakingContract`

Severity: Low Risk

Context: StakingRewards.sol#L160

Description: The `StakingContract` implements a privileged `recoverERC20` function that allows retrieving any ERC20 token (except the `stakingToken`), including the reward token. The rewards that users accrued can become unbacked and users might be unable to receive their rewards.

Recommendation: The functionality to retrieve reward tokens was likely implemented in case some rewards are not distributed which happens when `totalSupply = 0` during an active reward distribution. Therefore, we don't recommend removing this function but users should be aware that this functionality exists, and wards should ensure that, if reward tokens are retrieved, all past and future accrued rewards can still be paid out.

Maker: Acknowledged.

Cantina: Acknowledged.

3.2.5 Staking rewards are not distributed linearly

Severity: Low Risk

Context: VestedRewardsDistribution.sol#L162, StakingRewards.sol#L154

Description: The `DssVest` contract defines a linear reward schedule over the time period of `bgn` to `fin` (`bgn == clf` is enforced, no cliff). The `VestedRewardsDistribution` contract claims the vested rewards, transfers them to and notifies the `StakingContract`. The `StakingContract` then distributes the new rewards (and any remaining rewards of the last reward period) over a new period lasting `rewardsDuration` seconds. This makes the actual reward distributions heavily dependent on how often `VestedRewardsDistribution.distribute` is called and how long the `StakingContract.rewardDuration` is. This can lead to non-smooth reward distributions that are different from the linear vesting schedule defined in `DssVest`. As an extreme example, imagine nobody calls `distribute` in the first half of `bgn -> fin`, then `distribute` is called, and the entire first half of the rewards are distributed during the next `StakingRewards.rewardsDuration` time window. In addition, there's an incentive for reward-maximizing whales to call `distribute` as soon as they stake in the `StakingContract` and to prevent others from calling `distribute` before they staked.

Recommendation: To provide a reward distribution close to the linear schedule defined in `DssVest` using the current set of contracts, we recommend:

1. `VestedRewardsDistribution.distribute` should be called often. Consider having a keeper trigger this function if it hasn't been triggered for some time.
2. Choose a `StakingRewards.rewardsDuration` time period that isn't too small. This reduces the effect of delayed `distribute` calls.

3. The start of any `DssVest` reward vesting period `bgn` should ideally be after users can already stake in the `StakingContract`. Currently, `DssVest` allows defining vesting schedules that started up to 20 years in the past. The initial `distribute` call could distribute a large number of rewards to a few initial stakers over a potentially small `rewardsDuration`.

Maker: Thanks for that succinct, yet thorough comment. Your analysis is spot on.

This is something our team accepted as a limitation of the `StakingRewards`contract, and initially we thought about adding checks in `VestedRewardsDistribution` to help prevent misbehavior, but our simulations showed that the gas costs of trying to trick the system largely offset the gains obtained with it. So we decided to not add code to deal with it.

ChainSecurity audit report also corroborated our view in Section 7.2.

Regarding your recommendations:

1. Yes, there will be some automation in place, ensuring `distribute` is called often enough.
2. `rewardsDuration` is set to 7 days. This is the default from the original Synthetix contracts, and we couldn't find a justification to change it.
3. Yes, we are aware of that and since there will be a specific release date, the vesting streams will be configured using that one, which will be in the future.

Cantina: Acknowledged.

3.3 Informational

3.3.1 Broken links in various contracts

Severity: Informational

Context: StakingRewards.sol##L33, Pausable.sol#L22, Owned.sol#L19

Description: Within the staking contracts, the links that point to the Synthetix version of the contracts (as the current staking contracts were inspired from the Synthetix) are broken. The following links should be present:

- StakingRewards
- Pausable
- Owned

Recommendation: Consider replacing the broken links with the new ones.

Maker: Fixed in commit 775db2d5,

Cantina: Fixed.

3.3.2 Prefer `encodeCall` to `encodeWithSelector`

Severity: Informational

Context: SDAO.sol#L377

Description: The contract performs a low-level static-call to an EIP1271 contract. It is using `abi.encodeWithSelector` to encode the parameters.

Recommendation: Consider using `abi.encodeCall` as it type-checks that `digest` and `signature` are indeed the correct types defined as the arguments for `isValidSignature`. Wrong or missing argument types can be caught already during compile time, whereas `encodeWithSelector` does not give this guarantee.

```
staticcall(abi.encodeCall(IERC1271.isValidSignature, (digest, signature)))
```

Maker: To keep consistency with NGT and NST, this issue has been addressed in commit 26b03c38.

Cantina: Fixed.

3.3.3 Document why `unchecked` usage is safe

Severity: Informational

Context: SDAO.sol#L202, SDAO.sol#L236

Description: The contract uses `unchecked` arithmetic to save on gas. There are comments specifying why it's safe to perform an unchecked math operation for certain unchecked blocks, but not for all of them:

- SDAO.sol#L202, SDAO.sol#L236: Unchecked addition here is safe as the sum of all balances equals the `totalSupply`, and any overflow would have occurred already when increasing the `totalSupply`.

Recommendation: Consider commenting on all non-obvious unchecked code usage.

Maker: Fixed in commits f87cdb15 and 2a9215ec.

Cantina: Fixed.

3.3.4 Check that address contains code before calling it

Severity: Informational

Context: SDAO.sol#L376

Description: The contract performs a low-level `isValidSignature` static-call to an EIP1271 contract. This call is always performed if verifying the signature failed and it does not check if the address is a contract. The call will still be done for EOAs and precompiles. Calling EOAs will revert as they won't return the expected 32 bytes but certain precompiles will return data and could in theory match the expected magic number. It should not be possible to verify signatures on behalf of precompiles and this could lead to issues if a protocol uses low-address as an indirect burn address.

Additional info: Potential precompiles that could validate are:

1) identity (`0x4`) but it returns more than 32 bytes (because of the calldata encoding `4 + 32 + X > 32`) and the return data size check would fail.

2) `sha256 / ripemd160` (`0x2, 0x3`) return 32 bytes but when decoding to `bytes4`, in the current solidity versions it checks that only the 4 most-significant bytes are set, so one would need to find an entire 32 bytes collision to match `isValidSignature.selector || 28-zero-bytes`, which is infeasible.

Recommendation: While the current checks make it impossible to perform a successful verification on the current precompiles, we still recommend checking if the contract contains code before performing the call. This also saves gas on the path when an EOA signature verification fails as it would now skip the call that is already known to fail.

Maker: To keep consistency with NGT and NST, this issue has been addressed in commit 26b03c38.

Cantina: Fixed.

3.3.5 Deprecated non-standard ERC20 methods `increaseAllowance`/`decreaseAllowance`

Severity: Informational

Context: SDAO.sol#L271, SDAO.sol#L287

Description: The `increaseAllowance` and `decreaseAllowance` functions where adopted as being part of an ERC20 contract over the time as a measure to avoid front-running of `allowance`. As of our knowledge there is no demonstrated exploit involving the `allowance`, making these non-EIP functions obsolete in an ERC20 implementation. Openzeppelin and Solady already voted and planned to remove them from the standard `ERC20` implementation and maybe move them into an extension, if desired.

Recommendation: Consider removing these non-standard ERC20 functions from the `SDAO` token.

Maker: To keep consistency with NGT and NST, this issue has been addressed in commit 26b03c38.

Cantina: Fixed.

3.3.6 Add additional deployment verification

Severity: Informational

Context: CheckStakingRewardsDeploy.s.sol#L62

Description: The `CheckStakingRewardsDeploy` script verifies the deployments.

Recommendation: Consider adding the following additional checks:

1. Check that `dist.dssVest == vest` and `dist.stakingRewards == farm` to ensure that `dist` bridges the correct contracts. This then also implies that `dist.gem()` matches the reward tokens of `vest` and `farm` but it could also be checked again to be safe.
2. Check that `vest.res(vestId) == 1`, meaning that the `vestId` is restricted. It's important that only the `dist` contract can claim vests. Eventually, also check that `vest.mgr(vestId) == 0`.

Maker: For clarity, we changed the script to contain the checks for both suggestions. See commit de527011.

Cantina: Fixed.

3.3.7 Only standard ERC20 tokens supported

Severity: Informational

Context: VestedRewardsDistribution.sol#L161

Description: The staking reward contracts are generic and in theory allow any token to be the staking or reward token. However, the transfers in `VestedRewardsDistribution.distribute` and accounting in `StakingRewards` can fail when using non-standard tokens, like fee-on-transfer tokens or rebasing tokens.

Recommendation: Ensure no non-standard ERC20 tokens are used.

Maker: Acknowledged. All ERC20 contracts that will be used are 100% standard.

Cantina: Acknowledged.