By
Quit

Contract Review
Issue date
6/18/2024

Overview

The following is a review of the $DAM token and staking contracts. $DAM is an ERC20 token to be deployed on Blast L2. There will be six staking contracts, each modified Synthetix forks, each with an allocation of $DAM. Users who lock their $APE on Ethereum Mainnet will earn $DAM rewards on Blast according to their lock duration.

Contracts in scope for this review include:

- `DAM.sol`
- `StakingRewardHelper.sol`
- `StakingRewards.sol`
- `DamVault.sol`

This review is based on the mainnet deployment at `0x1ab1dA4DCBFc23b3bbEa589b0Eedb545D461AC3e`, and the commit SHA `51b0184ff00d4e5ed04e5d6ae2951d993b50daef`, and aims to identify security vulnerabilities, opportunities for gas optimization, and general best practice recommendations with regards to the contracts in scope. The review should not be considered an endorsement of the project, nor is it a guarantee of security. It is recommended to always source multiple independent audits in addition to testing extensively.

Findings

Outdated solidity version

Severity: Informational

`StakingRewards.sol` and `StakingRewardsHelper.sol` are written using solidity 0.5.16. Updating to a more recent version of solidity may offer gas savings and more consistency across contracts.

Gas is not set to claimable on Blast contracts

Severity: Informational

Blast redirects sequencer fees to the dapps that induced them, allowing smart contract developers to have an additional source of revenue. The default setting directs fees to the Blast sequences, but it is simple to set up your contract to receive these fees via a `claim` function.
See: https://docs.blast.io/building/guides/gas-fees#setting-gas-mode-to-claimable

`deposit` and `depositWithLock` can be combined

Severity: Informational

`deposit` can be removed, as `depositWithLock` can easily be adjusted to handle deposits with no lock period by changing the following two lines:

- Remove `require(lockYear > 0)`
- Preface `require(block.timestamp < lockEndTime[lockYear])` with `if (lockYear > 0)`

This change increases clarity and decreases duplicate code.

`deposit` does not follow Checks-Effects-Interactions pattern

Severity: Low

In `DamVault`, the deposit function makes several external calls within the internal `_deposit` call, which occurs before shares are added to the `_lockAmount` mapping. This could open up potential for reentrancy, however, the external calls have been vetted and there is no such. It is still recommended to follow CEI by adjusting the `_lockAmount` mapping before the internal `_deposit` call. Note that this also means that the condition checked on L149 needs to be adjusted.

Use unchecked math where possible

Severity: Gas Optimization

Many of the arithmetic operations have no risk of over or underflow, and can be safely marked unchecked in order to save gas.

- DamVault
  - L60, L72, L100-105, L130, L146, L153, L168-169*, L175, L193, L202

Use custom errors over require statements with strings

Severity: Gas Optimization

Strings are very expensive in solidity, which makes require(condition, error_string) less than ideal. Instead, opt to use custom errors which cuts down on deploy and revert costs.

Recommend using Solady or Solmate over OpenZeppelin where possible

Severity: Gas Optimization

Solady is a highly optimized Solidity utility library that can provide many of the same components that OZ does, with more efficiency.
https://github.com/Vectorized/solady

`_lockAmounts` are read twice during `withdraw`

Severity: Gas Optimization

When calling `withdraw`, the require statement on L86 checks `_lockAmounts` for a given user. It then calls the internal `_withdraw` function, which executes `_updateLockInfo`, once again reading all `_lockAmounts` for the same user. This redundancy can be avoided by removing the L86 `require` statement, and letting the arithmetic on L169 revert due to underflow if the user does not have enough shares to fill the withdrawal request. It's important in this case that L169 *not* be made unchecked, as suggested in "***Use unchecked math where possible***", because we are now relying on it to revert in case of underflow.

Centralization Risk
Severity: High
On blast, the staking contracts' owner retains the ability to swap out both the `helper` contract and the `messenger` contract. Either of these could potentially lead to manipulation of deposits or withdrawals to favor or disfavor particular users at the whim on the contract owner.

Lock period can be arbitrarily adjusted by contract owner
Severity: High
In DamVault, the contract owner has the ability to set a new lock end time for a lock year. This is meant to only allow decreasing lock times, however, there is an exception when the existing lock time is zero. This means that the contract owner can circumvent this requirement by first setting the lock time to zero (a decrease), and then extending it to whatever value they please.

Summary

No critical issues were uncovered as a result of this audit. The main concerns are two high severity issues that are classified as high impact, low probability. Both inherently boil down to a centralization risk - that being said, they are both avoidable, and adjustments would greatly benefit the robustness and trustlessness of the DAM system.

Several gas optimizations, and QA/informational findings were identified and while it is recommended that the team investigate and make adjustments as necessary, these are classified as low severity and it is not strictly necessary to make said adjustments.

This review assumes that the Synthetix contracts (https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol) that were forked for the $DAM ecosystem are free of vulnerabilities. It also assumes that the Cyan Wallet, and specifically the apecoin locking and unlocking logic works as intended and cannot be exploited.