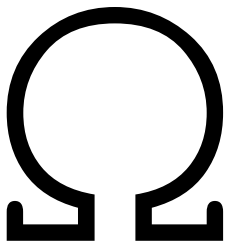
LSDAI

Final Audit Report

August 1, 2023



Team Omega

teamomega.eth.limo

Summary

The LSDai project has asked Team Omega to audit an upgrade of the LSDAI Token Contract.

We found **1 high severity issue** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **1** issue a “medium” - this is an issue we believe you should definitely address. In addition, **4** issues were classified as “low”, and **4** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Severity	**Number of issues**	**Number of resolved issues**
High	1	1
Medium	1	1
Low	4	4
Info	4	3

Scope of the Audit

We audited the code from the following repository and commit:

The scope of the audit concerns the changes made to the DXD Token. This code is currently developed in the following repository and branch:

```
https://github.com/lsdai/lsdai-contracts
```

We audited the following commit hash:

```
8a3893ed732df16763e7c5e2e65439eedb9599c1
```

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Resolution

The issues were addressed in the following commit:

```
427b543226886f4ab3d2e7759a76418c1ad66f5f
```

We reviewed the fixes and commented on the issues below

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of

the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Findings

General

G1. General state of the repository

The project's setup is not complete. The project mixes foundry with hardhat and npm - we recommend you choose either framework. There is no documentation for developers for installing or running the tests. Although there are github actions defined, continuous integration in github is not configured.
Recommendation: Clean up and fix the tests.

LSDAI.sol

L1. An attacker can steal most of the interest [high] [resolved]

The amount of shares the user will get on deposit depends on `_totalPooledDai`, which is updated when `rebase` is called. If `rebase` has not been called in the current block, a depositor can steal the interest from the other users by depositing and withdrawing shares.

One scenario is more or less as follows:

- `_totalPooledDai` is 100, `totalShares` are 100
- if we would call `rebase()`, `_totalPooledDai` would be 110
- An attacker deposits 100 dai and gets 100 shares, `totalPooledDAI` now is equal to 200
- The attacker calls `rebase()`, now `_totalPooledDai = 210`
- The attacker has 50% of the shares and withdraws 105 dai, i.e. half of the interest that rightfully should have gone to the shareholders from before the attacker joined

This can be done in a single transaction, and with a flash loan the attacker does not need any capital to deposit a large amount of DAI to take a higher proportion of the acquired interest.

A similar problem holds for the transfer of funds.

Recommendation: One way to solve this problem is to take inspiration from https://github.com/makerdao/sdai, which will call rebase before each deposit, withdrawal, and transfer. In this contract, the `balanceOf` function will report a value that is correct as long as the DSR (the DAI Savings Rate) does not change. However, the rebasing logic in that contract is much less costly, as it only needs to update the total balance, and does not need to calculate and transfer the fee.

One could also take a hint from the stETH contact, which has a (very costly) permissioned rebase function that is called by an oracle, and which is called once every 24 hours.

Severity: High

Resolution: The developers did not make any changes to address this issue, but noted that the contract charges a fee on withdrawal. If that fee (together with the gas and capital costs of the attack) is higher than the expected profit, the attack is not profitable. The profit for the attacker depends on the amount of interest that has accumulated since the last time `rebase` was called. So this means that the project can make sure the attack remains unfeasible by calling `rebase` on a regular basis, and by setting the `withdrawFee` high enough to offset the profit from the attack.

However, there is still an issue of fairness. Depositors that make their deposit receive a proportional part of the interest that accumulated in the time between the last rebase and the time of the deposit. This problem is mitigated, but not completely solved, by calling `rebase` on a regular basis.

L2. collectFees() burns shares of the sender instead of the feeRecipient [medium] [resolved]

The function `collectFees()` is meant to be called by the owner to collect the fees. But instead, what it will do is try to withdraw funds from the caller's account, for the amount of the balance of the `feeReceiver`.

Since the function is protected with `onlyOwner`, if the `feeReceiver` is not the owner, the claiming of fees will withdraw funds from the owner's account, and not the funds of the fees.
Recommendation: Implement the desired behavior.
Severity: Medium
Resolution: The issue was resolved as recommended.

L3. Implement sanity checks on fee and interestFee [low] [resolved]

In `setWithdrawalFee()`, no restrictions are put on the new value for the `withdrawalFee`. This means that the owner of the contract can set the `fee` at 100% (effectively taking the entire deposit of the user) or to a value higher than 100% (which will block withdrawals from users entirely, as these will revert). Similar considerations hold for `interestFee`, although with less dire consequences.
Recommendation: Enforce limits on the values for `withdrawalFee` and `interestFee`.
Severity: Low
Resolution: The issue was resolved as recommended.

L4. Remove SafeMath [low] [not resolved]

The contracts are compiled in solidity 0.8.20. The latest versions of solidity natively implement overflow checks for arithmetic operations, and so `SafeMath` is not needed any more.
Recommendation: Remove the `SafeMath` import and its use.
Severity: Low
Resolution: The issue was not resolved.

L5. remove unnecessary checks [low] [resolved]

On line 235, and again on line 249:

```
if (fee < 0) {
```

These checks are not necessary as `fee` is of the type `uint`, and so it can only be 0 or more.
Recommendation: Remove these needless checks on lines 235 and 249 to save some gas.
Severity: Low
Resolution: The issue was resolved as recommended.

L6. Remove unenforceable check for the cap [low] [resolved]

On line 33, in `setDepositCap`, it is checked whether the new cap value does not exceed the current DAI balance in the contract:

```
if (cap < _getTotalPooledDai()) {
```

This condition does not guarantee that the total amount of DAI in the pool is less than the `depositCap`, as the amount of DAI may grow on `rebase` when collecting interest.
In any case, it seems harmless for the `depositCap` being lower than the total amount of DAI in the contract, and there is a viable use case for setting the `depositCap` lower than that (for example, you could set the `depositCap` to a very low value if you for some reason want to disallow any new deposits and only allow withdrawals)
Recommendation: Remove this check.
Severity: Low
Resolution: The issue was resolved as recommended.

L7. Remove default values in proxy implementation [info] [resolved]

The contract defines values for `interestFee` and `withdrawalFee`. Because these values are stored in the implementation contract, they cannot be read by the proxy contract - the values that the proxy contract will use are those set by the `initialize` function.

```
uint256 public interestFee = 250; // 2.5%
uint256 public withdrawalFee = 1; // 0.01%
```

Recommendation: Remove these default values to avoid confusion (and save a minimal amount of gas on deployment).
Severity: Info
Resolution: The issue was resolved as recommended.

L8. potShares() can be declared external [info] [resolved]

The `potShares()` function is not called from anywhere else in the contract and can be declared `external`.
Recommendation: Mark the function external.

Severity: Info
Resolution: The issue was resolved as recommended.

L9. consider adding functions to manage allowances [info] [resolved]

It is common to add functions `increaseAllowance` and `decreaseAllowance` to ERC20 tokens to prevent potential abuse of the `approve` function. See for example OpenZeppelin's ERC20 implementation: https://docs.openzeppelin.com/contracts/2.x/api/token/erc20
Recommendation: implement `increaseAllowance` and `decreaseAllowance` functions
Severity: Info
Resolution: The issue was resolved as recommended.

L10. distribution of interest is not fixed [info] [not resolved]

Each time `rebase` is called, the balance of tokens of a user is updated with the accumulated interest since the last time `rebase` was called. This has some consequences on the distribution of interest. See also L1. This has as an effect that the amount of interest that gets distributed to a user depends on how many other users joined during a given rebase period.
As we've described in L1, if Ann is a depositor, and Bill joins during a rebase period, part of the interest that was accumulated on Ann's deposit during this period will go to Bill. Similarly, if Bill withdraws during a rebase period, he will not receive the interest accumulated since the last rebase - this interest will go to the depositors that remain.
Recommendation: Call rebase before each operation that changes the balance of a user, i.e. before `deposit`, `withdraw` and `transfer`.
Resolution: The issue was not resolved - the developers consider that if rebasing happens regularly, the small adjustments (losses or gains) in interest do not justify the transaction costs of rebasing before each action.