

Dai Savings Rate (DSR) Integration Review

Minimum Spanning Technologies

Version 1.1 – August 8, 2024

Prepared By
Paul Bottinelli
Javed Samuel

Prepared For
Sam Williams
Ivan Morozov
Christian Smorra

1 Executive Summary

Synopsis

In July 2024, Arweave engaged NCC Group's Cryptography Services team to conduct a short security review of a pull request introducing updates to the staking smart contract. The pull request implements support for depositing DAI through the DSR Manager[1] smart contract and generating yield on the deposits, using the *Dai Savings Rate* (DSR). The pull request also contained modifications resulting from the removal of outdated functionalities. The review was delivered remotely by one consultant over two person-days. A short retest was performed subsequently.

Scope

NCC Group's review was performed on the pull request "DAI Prebridge #1" located at https://github.com/Autonomous-Finance/ao-dai-prebridge/pull/1/files and merged at commit ID `2080f93`. The consultants also used public documentation on DSR[2] to complement their review.

Following the review, a pull request addressing the report findings was reviewed.

Limitations

Good coverage of the in-scope code was achieved overall. However, while the contracts were evaluated in isolation, they will be used as part of a more complex system which is not covered by this review. In addition, the implementation assumes that some entities are trusted, including the deposit token contract and the Distribution contract owner. It is paramount for the security of the system that these trusted entities are indeed honest.

Key Findings

The assessment uncovered one low severity issue:

- Finding "Potentially Incorrect Transfer Amount in `_withdraw()` Function", in which the transfer of an incorrect amount of funds in the `_withdraw()` function may result in inconsistencies in the total fund supply.

Additionally, the section Engagement Notes on the Dai Savings Rate (DSR) Integration Review contains a number of informal engagement notes.

After retesting, the NCC Group team found that the reported finding had been addressed.

Strategic Recommendations

Prioritize addressing the findings and numerous notes in the section Engagement Notes on the Dai Savings Rate (DSR) Integration Review. Consider using a static code analysis tool, such as `npm audit`, to catch outdated or vulnerable versions of the dependency packages. Running `npm audit` on the current repository revealed 65 vulnerabilities (33 moderate, 31 high, 1 critical) which is more than were uncovered in the previous review of the *Distribution* contract. Although some of these vulnerabilities might not be applicable to the smart contract's functionality, it is strongly recommended to keep abreast of the latest developments.

1. https://github.com/makerdao/dsr-manager/blob/master/src/DsrManager.sol
2. https://docs.makerdao.com/smart-contract-modules/proxy-module/dsr-manager-detailed-documentation



2 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Potentially Incorrect Transfer Amount in `_withdraw()` Function	Fixed	XUC	Low



3 Finding Details

Low

Potentially Incorrect Transfer Amount in `_withdraw()` Function

Overall Risk	Low	**Finding ID**	NCC-E014939-XUC
Impact	Medium	**Component**	Distribution
Exploitability	Low	**Category**	Data Validation
		Status	Fixed

Impact

The transfer of an incorrect amount of funds in the `_withdraw()` function may result in inconsistencies in the total fund supply.

Description

The `_withdraw()` function in *contracts/Distribution.sol* is used by users to withdraw their staked funds. The function takes an `amount_` as parameter and starts by performing a few validation steps, ensuring it is less than the `userData.deposited` amount (and upper-bounding it at that amount in case it is larger), see highlight on line 198. The function performs similar checks against the `daiBalance` tracked by the `daiDsrManager`, see second highlight below.

Once the appropriate `amount_` has been validated and computed, that quantity is then used to update local variables tracking the quantity of assets in the contract, including the variables `newDeposited_` on line 213 or `totalDepositedInPublicPools` on line 223, highlights 3 and 4 below.

```
189     function _withdraw(address user_, uint256 poolId_, uint256 amount_, bytes32
        ↳ arweaveAddress_) private {
190         require(amount_ > 0, "DS: nothing to withdraw");
191         Pool storage pool = pools[poolId_];
192         PoolData storage poolData = poolsData[poolId_];
193         UserData storage userData = usersData[user_][poolId_];
194
195         uint256 deposited_ = userData.deposited;
196         require(deposited_ > 0, "DS: user isn't staked");
197
198         if (amount_ > deposited_) {
199             amount_ = deposited_;
200         }
201
202         uint256 newDeposited_;
203         require(
204             (block.timestamp > userData.lastStake + pool.withdrawLockPeriodAfterStake),
205             "DS: pool withdraw is locked"
206         );
207
208         uint256 depositTokenContractBalance_ = daiDsrManager.daiBalance(address(this));
209         if (amount_ > depositTokenContractBalance_) {
210             amount_ = depositTokenContractBalance_;
211         }
212
213         newDeposited_ = deposited_ - amount_;
```




```
214         require(newDeposited_ >= pool.minimalStake || newDeposited_ == 0, "DS: invalid
            ↪ withdraw amount");
215
216         // Update pool data
217         poolData.lastUpdate = uint128(block.timestamp);
218         poolData.totalDeposited -= amount_;
219
220         // Update user data
221         userData.deposited = newDeposited_;
222
223         totalDepositedInPublicPools -= amount_;
224
225         uint256 dsrExitedAmount_ = _exitDsr(amount_);
226         IERC20(depositToken).safeTransfer(user_, dsrExitedAmount_);
227
228         emit UserWithdrawn(poolId_, user_, amount_, arweaveAddress_);
229     }
```

Figure 1: contracts/Distribution.sol

However, when transferring the funds back to the user at the end of the function (in the call to `IERC20(depositToken).safeTransfer(user_, dsrExitedAmount_);`, also highlighted above, the amount that is used in that call is `dsrExitedAmount_` and not `amount_`. These quantities *should* be equivalent, but the function `_exitDsr()` (which is called to obtain the `dsrExitedAmount_` variable) explicitly states that the amount withdrawn can be different than the requested amount. If that were the case, the amount transferred back to the user and the amount that is tracked within the Distribution contract would be different, which could result in inconsistencies in the total fund supply

Recommendation

Ensure the value transferred back to the user is exactly the same as the one used to compute the updated balances in the contract for the variables `poolData.totalDeposited`, `userData.deposited` and `totalDepositedInPublicPools`.

Location

contracts/Distribution.sol

Retest Results

2024-08-08 – Fixed

In pull request 3, the `_withdraw()` function in the Distribution contract was updated in a way that addresses the current finding (as well as some of the observations in the section Engagement Notes on the Dai Savings Rate (DSR) Integration Review). Specifically, the `amount_` variable is set to the return value of the call to `_exitDsr()` prior to any balance update, as follows:

```
187     require(amount_ <= depositTokenContractBalance_, "DS: amount is greater than contract
        ↪ balance");
188     uint256 dsrExitedAmount_ = _exitDsr(amount_);
189     amount_ = dsrExitedAmount_;
```

Figure 2: contracts/Distribution.sol

This follows the recommendation above. As such, this finding was marked “Fixed”.



4 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.



Rating	Description
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



5 Engagement Notes on the Dai Savings Rate (DSR) Integration Review

This section collects a number of notes and observations from the review that did not warrant stand-alone security findings, or that are not security related, but may nevertheless be of interest to the team at Arweave in order to improve the current posture of the code base.

Defence in Depth Opportunity in `_withdraw()` Function

The `_withdraw()` function in *contracts/Distribution.sol* is used by users to withdraw their staked funds. The function takes an `amount_` as parameter and starts by performing a few validation steps, for example ensuring that it is no larger than the amount originally deposited by the user. Following that, the function also checks that the requested amount is smaller than the total amount of tokens deposited and held in the `daiDsrManager` contract. If the amount requested is larger than the total deposited amount, the function sets the amount to the total, as can be seen in the code excerpt below.

```
function _withdraw(address user_, uint256 poolId_, uint256 amount_, bytes32 arweaveAddress_)
↳ private {
    // ...
    uint256 depositTokenContractBalance_ = daiDsrManager.daiBalance(address(this));
    if (amount_ > depositTokenContractBalance_) {
        amount_ = depositTokenContractBalance_;
    }
```

Arguably, if the function execution reaches that conditional statement with `amount_ > depositTokenContractBalance_`, something very wrong has happened. Indeed, this would mean that the amount of funds deposited by this user is *larger* than all funds tracked in the `daiDsrManager`. Consider replacing this conditional check by a `require` call.

Unnecessary Public Pool Attribute

The recent updates to the `Distribution` contract resulted in the deletion of all logic related to public pools, essentially assuming that private pools are no longer supported and thus all pools are public. However, the code base still provdes the ability to declare a pool as public. See the definition of the `Pool` structure in *contracts/interfaces/IDistribution.sol* for example:

```
    /**
     * The structure that stores the core pool's data.
     * @param withdrawLockPeriodAfterStake The period in seconds when the user can't withdraw
     ↳ his stake after staking.
     * @param minimalStake The minimal stake amount.
     * @param isPublic The flag that indicates if the pool is public.
     */
    struct Pool {
        uint128 withdrawLockPeriodAfterStake;
        uint256 minimalStake;
        bool isPublic;
    }
```

Note that non-public pools will trigger an error, due to the `poolPublic` modifier called upon pool creation, in *contracts/Distribution.sol*.

```
    modifier poolPublic(uint256 poolId_) {
        require(pools[poolId_].isPublic, "DS: pool isn't public");
        _;
    }
```

However, it might make sense to fully get rid of the `isPublic` attribute in pools, in order to simplify logic and force all pools to be public, by simply not even providing users with the ability to declare private pools.

Inaccurate Function Documentation

The documentation of the `_joinDsr()` and `_exitDsr()` functions is inaccurate; it describes actions that these functions *used to perform*, but no longer do. For example, the `_joinDsr()` documentation states that the function will ensure sufficient allowance and emit events, as highlighted below. The function no longer performs these actions.

```
    /**
     * @dev Deposits the specified amount of DAI into the Dai Savings Rate (DSR) system.
     * This function ensures sufficient allowance, performs the deposit, calculates the
     * actual amount deposited, and emits an event with the results.
     * @param amount_ The amount of DAI to deposit into DSR.
     * @return The actual amount of DAI deposited, which may differ slightly from the input
    ↳ amount.
     */
    function _joinDsr(uint256 amount_) private returns (uint256) {
```

This observation goes for the function `_exitDsr()` as well, which also states that an event is emitted, while a recent commit (with id `406fc6b`) replaced that with a `require` call.

```
    /**
     * @dev Withdraws the specified amount of DAI from the Dai Savings Rate (DSR) system.
     * This function performs the withdrawal, calculates the actual amount withdrawn,
     * and emits an event with the results.
     * @param amount_ The amount of DAI to withdraw from DSR.
     * @return The actual amount of DAI withdrawn, which may differ slightly from the input
    ↳ amount.
     */
    function _exitDsr(uint256 amount_) private returns (uint256) {
```

Issues with Negative Interest Rate

The DSR community resources highlight the fact that it's not *impossible* for the DSR to be negative:

> Are there any range limits to the DSR, can it be negative or zero?
>
> There are no hard limits to what the DSR can be set to. The exception being that it is not possible for the DSR to be set to a negative rate with the current code-base. In theory, it's possible for MKR governors to commission a rewrite of the code to be able to support negative rates. Whether this would make sense is another question. Negative rates on the DSR might not be effective, since users would simply withdraw their Dai.

In the interest of completeness, the NCC Group consultants wanted to point out that a negative interest rate would be incompatible with the current implementation. However, since this is a far-fetched scenario which would require a rewrite of the DSR smart contract code, this is most likely a non-issue.

Missing Return Value Check in `bridgeOverplus()` Function

The function `bridgeOverplus()` in *contracts/Distribution.sol* calculates the interest accrued on the balance in the DAI contract, "exits" the interest and then transfers all the funds in the

Distribution contract (including the interest called `overplus`) to the `aoDistributionWallet` address, see the code excerpt below.

```
    function bridgeOverplus() external {
        uint256 overplus_ = calculateOverplus();
        require(overplus_ > 0, "DS: overplus is zero");

        _exitDsr(overplus_);
        uint256 currentContractDaiBalance = IERC20(depositToken).balanceOf(address(this));
        require(currentContractDaiBalance > 0, "DS: contract has no DAI balance");

        IERC20(depositToken).safeTransfer(aoDistributionWallet, currentContractDaiBalance);

        emit OverplusBridged(overplus_);
    }
```

The function does not check the return value of the call to `_exitDsr(overplus_);`, highlighted above. The `_exitDsr()` function explicitly implies that the *actual* amount withdrawn may differ from the one passed as parameter, as also explained in the finding finding "Potentially Incorrect Transfer Amount in `_withdraw()` Function". The function *could* be updated to ensure the amount returned by the `_exitDsr()` is consistent with the one passed as parameter.

6 Contact Info

The team from NCC Group has the following primary members:

- Paul Bottinelli – Consultant
  paul.bottinelli@nccgroup.com
- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Javed Samuel – Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Minimum Spanning Technologies has the following primary members:

- Sam Williams
  sam@arweave.org
- Ivan Morozov
  ivan@dataos.so