PUBLIC

Code Assessment

of the dss-vest Smart Contracts

December 08, 2022

Produced for

by

Contents

1 Executive Summary

Dear Jean,

Thank you for trusting us to help Giry with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of dss-vest according to Scope to support you in forming an opinion on their security risks.

Giry implements a vesting plan for the participants of DAOs. The project is a fork of another well-audited project with a small number of additional features.

The most critical subjects covered in our audit are functional correctness and access control. We find that the project implementation is of a high quality and no severe issues were uncovered.

The general subjects covered are code complexity, use of uncommon language features, unit testing, documentation, specification, and gas efficiency. Security regarding all the aforementioned subjects is high with the exception of unit tests and the documentation which have not been updated to reflect the current state of the project. More specifically, the unit tests do not check for the correctness of the newly introduced features.

In summary, we find that the codebase provides a high level of security, but we strongly suggest to implement the missing unit tests.

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	6
• Code Corrected	1
• Acknowledged	5

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 dss-vest repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	15 Jun 2022	5c8c2e5141caf7f0936cc31dc900d73c1e796c58	Initial Version
2	29 Nov 2022	34bb53655286d715c95a08e752d3d8df143b4e8f	Fixes

For the solidity smart contracts, the compiler version `0.8.13` was chosen.

The following files were in scope:

- DssVest.sol

The project is a fork of dss-vest developed by MakerDAO commit: a4e42ab90ac21ec4294bb804cfa42e29f29466ab. In the current assessment, only the changes/extensions were reviewed. Moreover, Giry informed us that only the deriving `DssVestTransferrable` is going to be deployed.

2.1.1 Excluded from scope

Testing files, third-party libraries and any files not listed above were not in scope of this review. As Giry does not intend to use the DssVestMintable or DssVestSuckable contracts, they are also excluded from the scope of this review.

3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

dss-vest allows DAOs to create a vesting plan for participants. The base contract `DssVest` implements most of the functionality, while the three deriving contracts, `DssVestMintable`, `DssVestSuckable` and `DssVestTransferrable` implement various ways of paying out the vested tokens. However, Giry only intends to use the `DssVestTransferrable` implementation.

3.1 Vesting Plans

A vesting plan has several parameters:

- `usr`: The address of the recipient of the tokens supplied by the vesting plan.
- `bgn`: The timestamp marking the beginning of the vesting period.

- `clf`: The timestamp marking the cliff date of the vesting period.
- `fin`: The timestamp marking the end of the vesting period.
- `mgr`: The address of the manager of the vesting period.
- `res`: A boolean value marking whether or not the vesting plan is restricted.
- `tot`: The total reward amount for the vesting plan.
- `rxd`: The reward amount that the recipient has received so far.
- `bls`: A boolean value marking whether or not the vesting plan has been "blessed".

The reward is distributed linearly over the vesting period. The rate at which it is distributed is determined by `tot / (fin - bgn)`. The rewards start accumulating at the beginning of the vesting period but can only be claimed after the cliff date.

The manager may `yank` (terminate) the vesting plan at any point, in which case all rewards accumulated up to that point can still be claimed but any later rewards will not be distributed.

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

If a vesting plan is blessed, the manager of the plan can no longer terminate it. The vesting plan cannot be terminated unless the blessing is removed.

An admin of the system can set a global `cap` for the rate at which vesting plans distribute rewards. A vesting plan with a higher rate than the `cap` cannot be created.

3.2 Roles and Trust Model

There are three privileged roles in the system. They are the following:

1. Ward - A ward is an admin of the system. They may add or remove other wards, create new vesting plans and change the global `cap` for the distribution rate. Lastly, they can restrict, unrestrict, bless, unbless existing vesting plans or `yank` them. The role is considered trusted by the system and will not against the interest of its users.
2. Manager - A manager of a vesting plan is determined at its creation. The manager may `yank` the vesting plan, which stops the rewards from accumulating any further. This role is considered trusted by the system that it will not act maliciously.
3. Recipient - The recipient of a vesting plan has certain privileges as well. They may restrict or unrestrict their vesting plan. If the vesting plan is restricted, they are the only one who can trigger distribution of their rewards. Finally, they can `move` the vesting plan, essentially transferring it to another recipient.
4. Everyone else - If a vesting plan is not restricted, anyone can trigger the distribution of rewards (but they still go to the recipient of the vesting plan).

4 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.

5 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.

6 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. 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	5

6.1 Redundant Calculation in _yank

Design Low Version 1 Acknowledged

When `_yank` is called, it determines how much reward can still be claimed by the recipient. If the new `_end` of the vesting plan is before the beginning or the cliff date of the vesting plan, the entire reward amount is cancelled. Otherwise, the following calculation is done:

```
awards[_id].tot = toUint128(
                    add(
                        unpaid(_end, _award.bgn, _award.clf, _award.fin, _award.tot, _award.rxd),
                        _award.rxd
                    )
                );
```

`unpaid` calculates the following result:

```
function unpaid(uint256 _time, uint48 _bgn, uint48 _clf, uint48 _fin, uint128 _tot, uint128 _rxd) internal pure returns (uint256 amt) {
    amt = _time < _clf ? 0 : sub(accrued(_time, _bgn, _fin, _tot), _rxd);
}
```

As we know that `_end` is after the beginning and cliff date of the vesting plan, this calculation can be simplified. The addition of `_rxd` in `_yank` and subtraction of `_rxd` in `unpaid` cancel out, so the final result is simply: `toUint128( accrued(_end, _bgn, _fin, _tot) )`.

Acknowledged:

Giry chooses not to modify the code out of caution, as the original dss-vest code has been the subject of extensive formal checking and is battle-tested.

6.2 Redundant Overflow Checks

Design Low Version 1 Acknowledged

As of Solidity version 0.8.0, the compiler automatically inserts overflow checks when doing integer arithmetic. As such, the `add`, `sub` and `mul` functions are redundant.

Additionally, the following require statement in `_create` can also be considered redundant:

```
require(ids < type(uint256).max, "DssVest/ids-overflow");
```

This statement makes sure that the increase of the id (`id = ++id`) in the following line will not overflow.

Acknowledged:

Giry chooses not to modify the code out of caution, as the original dss-vest code has been the subject of extensive formal checking and is battle-tested.

6.3 Redundant Storage Load in _yank

Design Low Version 1 Acknowledged

The `_yank` function loads `awards[_id]` from storage multiple times:

```
function _yank(uint256 _id, uint256 _end) internal lock {
    require(wards[msg.sender] == 1 || awards[_id].mgr == msg.sender, "DssVest/not-authorized");
    Award memory _award = awards[_id];
```

It is important to note that since the Berlin hardfork, the memory slot of `awards[_id]` is considered warm, thus the benefit of such optimization is limited.

See more here https://eips.ethereum.org/EIPS/eip-2929.

Acknowledged:

Giry chooses not to modify the code out of caution, as the original dss-vest code has been the subject of extensive formal checking and is battle-tested.

6.4 Reentrancy Lock Can Be Cheaper

Design Low Version 1 Acknowledged

Using different values for the `locked` variable results in cheaper transactions overall. Setting a storage variable from 0 to 1, but then resetting it back to 0 costs ~20112 gas but causes 19900 gas to be refunded. However, the total refund amount a transaction is eligible for is limited to a fraction of the total gas expended by the transaction. Hence, for simple transactions it is likely that not the entire gas refund will be received.

Instead, one could initialize the `locked` variable with a value of 1 in the constructor. Then, at the start of a transaction set it to 0 for the reentrancy lock and setting it back to 1 at the end. This costs ~3012 gas, but refunds 2800. Assuming a complete refund, this costs the same amount of gas per transaction, but is more likely to be completely refunded as the total refund amount is smaller.

For completeness, we should note that a similar practice is utilized by the OpenZeppelin library. In this case, the values 1 and 2 are used instead of values 1 and 0. The efficiency of this approach is similar.

For more information about gas refunds and the exact refund amounts, see https://eips.ethereum.org/EIPS/eip-3529.

Acknowledged:

Giry chooses not to modify the code out of caution, as the original dss-vest code has been the subject of extensive formal checking and is battle-tested.

6.5 Unnecessary Calculation in `accrued`

Design | Low | Version 1 | Acknowledged

In the `accrued` function, unnecessary calculations are performed in the case where `_time == _bgn`. In this case, the result will be `0`. As such, the first if condition could be modified to be `_time <= _bgn` in order to save gas in this case.

```
function accrued(uint256 _time, uint48 _bgn, uint48 _fin, uint128 _tot) internal pure returns (uint256 amt) {
    if (_time < _bgn) {
        amt = 0;
    } else if (_time >= _fin) {
        amt = _tot;
    } else {
        amt = mul(_tot, sub(_time, _bgn)) / sub(_fin, _bgn); // 0 <= amt < _award.tot
    }
}
```

Acknowledged:

Giry chooses not to modify the code out of caution, as the original dss-vest code has been the subject of extensive formal checking and is battle-tested.

7 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

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

- Code Duplication Code Corrected

7.1 Code Duplication

Design Low Version 1 Code Corrected

The `_bless` and `unbless` functions check for the condition `wards[msg.sender] == 1`. This is the same condition as is enforced by the `auth` modifier. In order to reduce code duplication, the auth modifier could be applied to these functions instead.

Code corrected:

The condition check was removed and replaced with the `auth` modifier.