PUBLIC

Code Assessment

of the Enso Vesting Smart Contracts

October 6, 2021

Produced for

by

Contents

1 Executive Summary

Dear Sir or Madam,

First and foremost we would like to thank EnsoLabs for giving us the opportunity to assess the current state of their Enso Vesting system. This document outlines the findings, limitations, and methodology of our assessment.

Initially, our code assessment resulted in a number of low severity findings and one medium severity issue. After the submission of the intermediate report, all findings have been resolved. These have been marked accordingly and can be found in the Resolved Findings section.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

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	1
• Code Corrected	1
Low-Severity Findings	4
• Code Corrected	4

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 Enso Vesting 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	September 18 2021	81041dc31b200d8864d6abaa105b093478d04018	Initial Version
2	October 5 2021	0963a7ba5ecacf7b592abfc3c332a63a441a717a	Version with fixes

For the solidity smart contracts, the compiler version `0.8.2` was chosen. File `contracts/Vesting.sol` is the only contract that was part of this assessment.

2.1.1 Excluded from scope

Contracts in the `contracts/mocks` folder of repository are excluded from scope. Imported libraries are assumed to be working correctly according to their specification.

2.2 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.

The `Vesting` contract implements the logic of vesting for the ENSO tokens. The contract owner registers users to the system by specifying the allocated amount of ENSO token (ERC20 token) of each user and their respective unlock time. Registered users can claim a portion of their ENSO token at any time and they should get their respective allocated amount of ENSO token after the lock time if they don't violate any prior agreement.

2.2.1 Claiming system

Each user has properties that are involved in `Vesting.getClaimable()` function:

- `last`: last time the vester claimed; initialized to the start time of the vesting system at the beginning
- `unlock`: time to unlock the user.
- `allocated`: total token allocated.
- `claimed`: total token claimed.

A registered user can call the `Vesting.claim()` function at any time. However, two different scenarios can happen:

The user is still locked: In this case, the claimable amount (result from `Vesting.getClaimable()` function) is added to the property `claimed`, the user receives the claimable amount of ENSO token and the `last` property is updated.

The user is unlocked: The user gets the remaining amount of ENSO token and then gets deleted from the users list.

2.2.2 Trust Model

The owner of the contract stored in `owner` is considered to be trusted by the system and assumed to act in a non malicious way.

3 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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 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 severities. 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.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of 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	0

6 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	1

- Mismatch Between Vesting.register() and onlyRegistered() Modifier Code Corrected

Low-Severity Findings	4

- Division Before Multiplication Code Corrected
- Gas Optimizations Code Corrected
- Outdated Compiler Code Corrected
- Sanity Checks Missing Code Corrected

6.1 Mismatch Between `Vesting.register()` and `onlyRegistered()` Modifier

Design Medium Version 1 Code Corrected

The vesting system accepts registering users with 0 allocated amount of tokens. However, the `onlyRegistered()` modifier requires that the allocated amount is greater than 0:

```
  modifier onlyRegistered(address _vester) {
    require(getAllocated(_vester) > 0, "Vesting#onlyRegistered: not registered");
    _;
}
```

Hence, registered users with 0 allocated amounts stay forever in the contract storage as they cannot call the `Vesting.call()` and the owner cannot unregister them because these functions use the `onlyRegistered()` modifier.

Code correct :

A boolean field `registrered` has been added to the `Vest` struct. It is now used to determine whether user is registered or not in `onlyRegistered` modifier.

6.2 Division Before Multiplication

Design Low Version 1 Code Corrected

In `Vesting.getClaimable` a multiplication is performed after a division:

```
function getClaimable(address _vester)
...
        (getAllocated(_vester) / (getUnlock(_vester) - start))
            * (block.timestamp - getLast(_vester))
...
```

Division can lead to rounding errors which are magnified by the multiplication that takes place later. Thus, multiplications should happen before divisions.

Code corrected :

Multiplication happens before division.

6.3 Gas Optimizations

Design | Low | Version 1 | Code Corrected

There are multiple gas inefficiencies in the `Vesting` contract:

1. `start` and `enso_token` could be immutable.
2. `batchRegister` and `batchUnregister` check whether the caller is the owner of the contract multiple times due to repetitive calls to the `onlyOwner` modifier.

Code corrected :

1. `start` and `enso_token` are now `immutable`
2. Two new internal functions `_register` and `_unregister` have been added. The logic of `register` and `unregister` has been moved to those functions. Now `register/batchRegister` and `unregister/batchUnregister` simply call those new functions and check only once for `onlyOwner`.

6.4 Outdated Compiler

Design | Low | Version 1 | Code Corrected

The `solc` version is fixed in the hardhat configuration to version `0.8.2`. This version is outdated with the most recent one being `0.8.7`.

Code corrected :

The `solc` version in hardhat configuration has been updated to `0.8.9`.

6.5 Sanity Checks Missing

Design | Low | Version 1 | Code Corrected

Sanity checks for the arguments of the variables are missing:

1. `unlock` is not checked to be greater than `start`.
2. `allocated` is not checked to be greater than 0.

Code corrected :

1. `unlock > start` check added
2. 0 check added for `allocated`

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 `getClaimable` Returns Wrong Value

Note Version 1

`Vesting.getClaimable` is responsible for returning the amount that is claimable by the vester. However, `getClaimable` would return a positive result even after all the remaining amount has been claimed. External parties might misinterpret the intention of the function in the absence of documentation.