EMDX Audit

Token contracts

November 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the EMDX project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The contracts audited are from the EMDX repository at https://github.com/emdx-dex/token-contracts.git. The audit is based on the commit `2a376848d3d0a3604f8c6ca702fe4718019d1b3d`. Fixes were made and rechecked based on the commit `2ffe6fa5eba3759275a65f855ae26e5954f4cd6c`.

Contracts

The audited contracts are:

- `contracts/EMDXToken.sol`: Simple ERC20 token
- `contracts/Vesting.sol`: Vesting contract

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Findings and Fixes

ID	Title	Severity	Status
ME-01	Denial of Service in Vesting.updateScore()	Medium	Mitigated
ME-02	Possible Overflow in Vesting.updateScore()	Medium	Fixed
EN-01	SafeMath Libraries Removal	Enhancement	Acknowledged

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical Severity

No issues found

Medium Severity

ME-01 Denial of Service in Vesting.updateScore()

The update score function will not work if any of the transfers in line 172 fails, reverting the transaction.

Recommendation

Use the "Favor pull over push for external calls" pattern to solve this problem. See https://eth.wiki/howto/smart-contract-safety#favor-pull-over-push-for-external-calls for details.

Solution

The development team informed us that they decided that the "Favor pull over push for external calls" pattern is not required in this instance because:

- less than 10 transfers will be made on each call.
- all the funds required will always be available.
- the token to be used is `EMDXToken` (in the same git repository and also part of this audit), making it not possible to trigger the denial of service.

ME-02 Possible Overflow in Vesting.updateScore()

An overflow may occur if `totalAmount` is bigger than $2^{256}/99$. This overflow will block the `Vesting.updateScore()` function on every invocation, stopping the recovery of any token owned by the `Vesting` contract with a smaller or similar score to the one passed in the `_newScore` variable.

Hereunder are the lines 157-165 of `Vesting.sol`. In bold the expression where the overflow will be triggered.

```
LockVesting memory lock = locks[beneficiaries[i]];
// calculate already vested percentage
uint256 remainingAmount = lock.totalAmount.sub(
    lock.releasedAmount
);
// calculate amount to be vested
uint256 releasableAmount = _newScore.mul(remainingAmount).div(
    100
);
```

Recommendation

Add a require statement in the `Vesting.grantVesting()` function checking that the `_amount` does not exceed $2^{256}/99$.

Solution

The issue was fixed by the development team.

Minor Severity

No issues found

Enhancements

EN-01 SafeMath Libraries Removal

Given that the solidity version is >= 0.8 there is no need to use `SafeMath` libraries. Usage of "`@openzeppelin/contracts/utils/math/SafeMath.sol`" in Vesting.sol is not required. See https://docs.soliditylang.org/en/v0.8.9/080-breaking-changes.html#how-to-update-your-code for details (section "How to update your code").

Solution

The development team informed us that they decided not to do this enhancement because they want to minimize the code changes to be made.

Other considerations

Vesting schedule

While in standard vesting schedules the amount that can be vested on each period is decided when the vesting is started, in the `Vesting` contract the tokens awarded on each period are decided by a score passed by the oracle using the `Vesting.updateScore()` function in lines 131-185 of `Vesting.sol`, when each vesting period starts. This includes the option to set the score as 0 or not calling the `Vesting.updateScore()`, which would not award any tokens for the period.

Conclusion

We found 2 medium-severity issues. Also an enhancement was recommended.

One of the medium-severity issues was fixed and the other was mitigated. The enhancement was not considered necessary by the development team.