OZYS VestingToken Security Analysis Report

Prepared by

78ResearchLab

RedSpider

Sep 13, 2024

TABLE OF CONTENTS

PROJECT OVERALL

About Project

This project involves the creation of a token that allows for customizable vesting models for specific addresses. Each address can be assigned a unique vesting type, with options including Linear and Halflife (decay) models. The token will inherit these features and be deployed as the governance token for a decentralized exchange (DEX) on the Silicon chain. By default, the token operates under the Halflife vesting model, but it provides the flexibility to assign different vesting schedules to individual addresses based on specific requirements.

Target Summary

Name	VestingToken
Website	https://ozys.io/
Repository	https://git.ozys.work/silicon/swap/vesting-token-contract
Commit	a90e7bef5b983e5477fabbc205ad765d3feeffc3
Network	Silicon
Languages	Solidity
Method	Source code auditing
Timeline	Aug 28, 2024 ~ Aug 30, 2024

SCOPE

Scope mainly consists of two contracts.

The VestingToken contract is a customizable token that allows specific addresses to have different vesting models, including Linear and Halflife models. It manages the vesting process, token minting, and allows burning of locked tokens based on each address's vesting configuration.

The VestingTokenForSwap contract extends the functionality of VestingToken, designed for use in a decentralized exchange (DEX) on the Silicon chain. It includes additional functionality for mining rewards, allowing tokens to be mined over time based on a halflife model and distributed as rewards via the `sendReward` function.

Source code

Name	commit
VestingToken	a90e7bef5b983e5477fabbc205ad765d3feeffc3

```
├── address
│   └── siliconTestnet
│       └── core.json
├── contracts
│   ├── interfaces
│   │   ├── IFactory.sol
│   │   └── IGovernance.sol
│   ├── TestProxy.sol
│   ├── VestingTokenForSwap.sol
│   └── VestingToken.sol
├── hardhat.config.js
├── package.json
├── README.md
├── scripts
│   ├── 1_deploy.js
│   └── 2_deploy_swap.js
└── test
    ├── deploy.js
    └── utils.js
```

RISK CLASSIFICATION

Severity

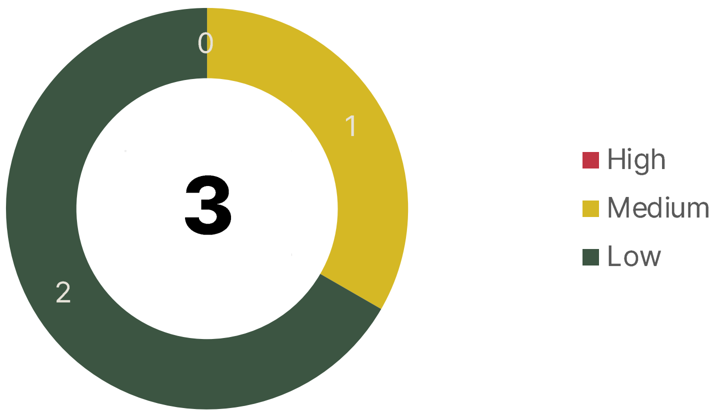
Our risk classification is based on Severity Categorization of code4ena.

> High ●
>
> Assets can be stolen, lost, compromised directly or indirectly via a valid attack path
> (e.g. Malicious Input Handling, Escalation of privileges, Arithmetic).

> Medium ●
>
> Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> Low ●
>
> Assets are not at risk. User mistake, misuse of privileges, governance risk fall under this grade.

FINDINGS BREAKDOWN



Severity	Acknowledged	fixed	Total
● High	0	0	0
● Medium	1	0	1
● Low	0	2	2
			3

* Fixed : Risk is fixed by Ozys.
* Acknowledged : Ozys has recognized the risk but has not addressed it, as it poses only a minor impact.

FINDINGS

● MEDIUM

M-01. VestingToken : Wrong caculation for HALFLIFE within _getTotalVestingAmount Acknowledged

IMPACT

When processing the HALFLIFE model in `_getTotalVestingAmount`, incorrect calculations of `amount` may result in locked amounts remaining in accounts.

DESCRIPTION

```
function _getTotalVestingAmount(Config memory c) internal pure returns (uint256 amount)
{
    if (c.model == Model.LINEAR) {
        amount = uint256(c.n3) * (c.n2 - c.n1);
    } else if (c.model == Model.HALFLIFE) {
        amount = uint256(c.n3) * 2 * c.n1;
    }
}
```

File 1 : VestingToken.sol Function: _getTotalVestingAmount :

For example, with the following values:

- halflife (n1): 10
- starttime (n2): 0
- amountPerSecond (n3): 4
- block.timestamp: 40

The function currently returns 80 as the total vesting amount.

However, the actual total amount of tokens released over time based on the halflife model would be 4 * 10 + 2 * 10 + 1 * 10, resulting in 70 tokens.

This discrepancy occurs because amount of token cannot drop below 1, leading to a calculation error. Due to this error, functions like `_getLockedAmount` or `getConfigs` may not behave as expected, causing the contract to fail to operate correctly in the long term.

RECOMMENDATIONS

Adjust the formula to account for value that drop below 1 during the halflife calculation, ensuring accurate token vesting calculations.

STATUS

Acknowledged

Ozys :

Hello,

Regarding the halflife and amountPerSecond values, if they are set to sufficiently large values, the error may be negligible. I expect the discrepancy to be around n * halflife.

For tokens with 18 decimals, the amountPerSecond is likely to be set at a minimum of 1e9, and this would allow for approximately 30 halvings before any noticeable error occurs.

For example, if the halflife is set to 1 year, then halflife = 31,536,000.

Given the practical values that would be used, the error would likely be less than 1e9, which seems insignificant, so I believe it may be acceptable to proceed while tolerating this margin of error.

Please share your thoughts. Thank you.

● LOW

L-01. VestingToken, VestingTokenForSwap : The loop within the _vestingMined may execute more iterations than necessary Fixed

IMPACT

When the vesting model is set to HALFLIFE, for loop in the _vestingMined function doesn't stop even when amountPerSecond becomes 0, resulting in excessive gas consumption. Similarly, the mined function in VestingTokenForSwap has the same issue, leading to unnecessary gas costs.

DESCRIPTION

The _vestingMined function calculates the number of tokens mined based on the vesting model configuration.

```
function _vestingMined(Config memory c) internal view returns (uint256 res) {
    if (c.model == Model.LINEAR) {
        if (block.timestamp < c.n1) {
            res = 0;
        } else if (block.timestamp < c.n2) {
            res = uint256(c.n3) * (block.timestamp - c.n1);
        } else {
            res = uint256(c.n3) * (c.n2 - c.n1);
        }
    } else if (c.model == Model.HALFLIFE) {
        uint256 startTime = c.n2;
        if (block.timestamp < startTime) return 0;

        uint256 level = (block.timestamp - c.n2 + 1) / c.n1;
        uint256 amountPerSecond = c.n3;

        for (uint256 i = 0; i < level; i++) {
            if (startTime + c.n1 > block.timestamp) break;

            res = res + (amountPerSecond * c.n1);
            startTime = startTime + c.n1;
            amountPerSecond = amountPerSecond / 2;
        }
        res = res + amountPerSecond * (block.timestamp - startTime + 1);
    }
}
```

File 2 : VestingToken.sol Function: _vestingMined

However, even when amountPerSecond reaches 0 after enough time has passed, the level value can continue to increase, leading to unnecessary iterations of the for loop. This can result in excessive gas consumption for users.

The same issue occurs in the `mined` function of VestingTokenForSwap, leading to additional gas costs for users.

RECOMMENDATIONS

Break the `for` loop when `amountPerSecond` becomes 0 to prevent unnecessary iterations.

STATUS Fixed

Ozys :

Hello,

We have added the following condition to both the `mined()` and `_vestingMined()` functions:

```
if (amountPerSecond == 0) break;
```

In fact, since both `amountPerSecond` and `halflife` are set to relatively large values, it is expected that `amountPerSecond` will not reach zero for at least several decades.
thank you.

Fixed in commit `be178eb0c17d1f1717906905982ce623eca8ef05`.

L-02. VestingTokenForSwap : Recommendation to Call mintWithConfig in initializeMining Fixed

IMPACT

Depending on the values set in `initializeMining`, there may be a discrepancy between the number of tokens returned by the `mined` function and the actual tokens minted through `mintWithConfig`, potentially causing transfer failures during the `sendReward` process.

DESCRIPTION

In the VestingTokenForSwap contract, the `mined` function returns the number of tokens mined so far based on the values set in `initializeMining`. However, the `mined` function only calculates and returns the amount but does not actually mint the tokens.

The only way to mint tokens in the VestingTokenForSwap contract is by calling the `mintWithConfig` function in the VestingToken contract. To mint tokens via `mintWithConfig`, you need to calculate separate configuration values and pass them as parameters, independent of the values set in `initializeMining`.

If the configuration values are miscalculated during this process, discrepancies in the number of tokens available for transfer during `sendReward` may arise, potentially causing the contract to malfunction.

RECOMMENDATIONS

When calling `initializeMining`, also call `mintWithConfig` to ensure that the tokens are minted appropriately.

STATUS Fixed

Ozys :

Hello,

As per your suggestion, to avoid the risk of misconfiguration, I have modified the contract so that `initializeMining` now calls `mintWithConfig` directly.

Fixed in commit `f691cfc4b25f2cbc0ddd7c83975d45dc46a5b2c3`.

Thank you.

ABOUT 78ResearchLab

78ResearchLab is a offensive security corporation offering security auditing, penetration testing, education to enterprises, national organizations, and laboratories with the goal of making safe and convenience digital world. We have our own proprietary technology from system/security analysis and projects on various industries. We are working with the top technical experts who have won prizes in global Realword Hacking Competition/CTF, reported numerous security vulnerabilities, and have 10 years of experience in the information security.
Learn more about us at https://www.78researchlab.com/.

ABOUT RED SPIDER

Red Spider offers cyber security research and auditing service with our new R&D technologies and customized solution in IoT, OS, Web3.
Learn more about red spider at https://www.78researchlab.com/redSpider.html.