Smart Contract Audit Report

July, 2023

DEFIMOON PROJECT

Audit and Development

CONTACTS

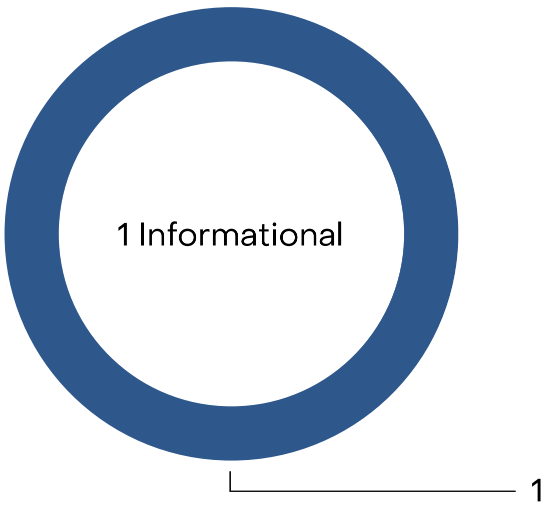
defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

4 July 2023

This audit report was prepared by DefiMoon for IPAD.

Audit information

Description	Tokens vesting contract
Audited files	IVesting.sol, Vesting.sol
Timeline	8 June - 4 July 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/pycrash/vesting-contract/tree/d40392a51e7496d077b83cf320b4df445c832894
Reaudit Source code	https://github.com/pycrash/vesting-contract/tree/0827d5824f368214a92407db51b0b5041c4b0bae
Status	Passed



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

No major problems in the contract design logic found at reaudit.

Initial engagement has discovered the following issues:

The contract contains design logic that cannot guarantee reliability to users.

The contract contains problems with linear vesting calculations.

The contract contains insufficient number of input argument checks.

The contract uses the deprecated SafeMath library, which is redundant. Using SafeMath only complicates the code, makes it more expensive in terms of gas to deploy and use.

The contract lacks at least a minimum number of events. Please use events for at least all major functions. Events can provide a lot of useful information in the future, help you collect statistics and programmatically track transactions to a contract.

Summary of findings

ID	Description	Severity	Status
DFM-1	No guarantees for users	Medium Risk	Resolved
DFM-2	Linear percentages may not be calculated correctly	Medium Risk	Resolved
DFM-3	Insufficient checks when adding addresses	Low Risk	Resolved
DFM-4	Insufficient checks when adding vesting	Low Risk	Resolved
DFM-5	Potential loss of owner	Low Risk	Resolved
DFM-6	Potentially incorrect late vesting logic	Informational	Open
DFM-7	Potentially reentrancy	Informational	Resolved
DFM-8	Using safeTransfer	Informational	Resolved
DFM-9	Outdated and unused libraries	Informational	Resolved
DFM-10	More secure calculations	Informational	Resolved

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Passed
Function input parameters lack of check	Passed
Function input parameters check bypass	Passed
Function access control lacks management	Passed
Critical operation lacks event log	Passed
Human/contract checks bypass	Passed
Random number generation/use vulnerability	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Assert () misuse	Passed
High consumption 'for/while' loop	Passed
High consumption 'storage' storage	Passed
"Out of Gas" Attack	Passed

Findings

DFM-1 «No guarantees for users»

Status: Resolved

Severity: Medium Risk

Description: The design of the contract is designed in such a way that it does not contain any function for replenishing the reserves of tokens, which cannot guarantee users the reliability of vesting.

Even if we take into account that tokens can be directly transferred to the contract address, then when adding addresses, there are no checks that guarantee that there are enough tokens on the balance for this or that user.

In addition, the contract contains the revoke function, which allows you to withdraw all tokens from all available vestings, including those that have not yet been completed or not all users claim tokens.

Recommendation: It is worth using implementations that can provide users with guarantees that they will be able to get their funds and remove the revoke function.

You can use one of the following approaches:

```
function addWalletAddress(
    uint256 _vid,
    address[] memory _addresses,
    uint256[] memory _amounts
) internal onlyOwner {
    uint256 l = _addresses.length;
    // ...
    Vesting storage vesting = vestingInfo[_vid];
    uint256 toTransfer;
    for (uint256 i; i < l; ) {
        // ...
        toTransfer += _amounts[i];
        // ...
        unchecked { ++i; }
    }

    if (toTransfer > 0) {
        IERC20(vesting.tokenAddress).safeTransferFrom(msg.sender, address(this),
toTransfer);
    }
    // ...
}
```

or

```
struct Vesting {
    // ...
    uint256 treasury;
    // ...
}

function replenishTheTreasury(
    uint256 _vid,
    uint256 _amount
```

```
) external {
    require(_amount > 0, "Vesting: Incorrect amount");
    Vesting storage vesting = vestingInfo[_vid];
    IERC20(vesting.tokenAddress).safeTransferFrom(msg.sender, address(this),
_amount);
    vesting.treasury += _amount;
}

function addWalletAddress(
    uint256 _vid,
    address[] memory _addresses,
    uint256[] memory _amounts
) internal onlyOwner {
    uint256 l = _addresses.length;
    // ...
    Vesting storage vesting = vestingInfo[_vid];
    uint256 toAdd;
    for (uint256 i; i < l; ) {
        // ...
        toAdd += _amounts[i];
        // ...
        unchecked { ++i; }
    }

    require(vesting.treasury >= toAdd, "Vesting: Not enough tokens in the
treasury");
    vesting.treasury -= toAdd;
    // ...
}
```

DFM-2 «Linear percentages may not be calculated correctly»

Status: Resolved

Severity: Medium Risk

Description: The implementation for linear vesting includes the presence of TGE in you, but it is incorrectly handled in the linear interest calculation function.

Example (multipliers are ignored):

interval: 2
unlocks: [10, 20, 30]
percentages: [40]

If the vesting interval is 2, then the tokens will be unlocked at timestamps 20, 22, 24, 26, 28, 30 (6 times), which means that 10% ((100 - 40) / 6) should be unlocked each time.
If we take the current timestamp as 22, then the result should be: 40% for TGE and 20% for reaching the timestamp at 22, for a total of 60%.
But following your implementation, the percentage of TGE is ignored when the token unlock phase starts:

```
} else {
    uint256 releasesCount = vesting
        .unlocks[vesting.unlocks.length - 1]
        .sub(vesting.unlocks[1]);
    // uint256 releasesCount = 30 - 20 = 10
    releasesCount = releasesCount.div(vesting.interval);
    // releasesCount = 10 / 2 = 5
    releasesCount = releasesCount.add(1);
    // releasesCount = 5 + 1 = 6

    uint256 timeAfterCliff = currentTimeStamp.sub(vesting.unlocks[1]);
    // uint256 timeAfterCliff = 22 - 20 = 2
    uint256 availableReleases = timeAfterCliff
        .div(vesting.interval)
        .add(1);
    // uint256 availableReleases = 2 / 2 + 1 = 2
    availableReleases = availableReleases.mul(1e20);
    // availableReleases = 2 * 100 = 200

    return availableReleases.div(releasesCount);
    // return 200 / 6 = 33%
}
```

Recommendation: The correct solution would be to consider TGE in all subsequent stages of vesting, for example:

```
} else {
    uint256 percentage = vesting.percentages[0];
    // percentage = 40
    uint256 releasesCount = vesting.unlocks[vesting.unlocks.length - 1] -
vesting.unlocks[1];
    // uint256 releasesCount = 30 - 20 = 10
    releasesCount = releasesCount / vesting.interval + 1;
    // releasesCount = 10 / 2 + 1 = 6
    uint256 intervalPercent = (1e20 - percentage) / releasesCount;
    // uint256 intervalPercent = (100 - 40) / 6 = 10

    uint256 timeAfterCliff = currentTimeStamp.sub(vesting.unlocks[1]);
    // uint256 timeAfterCliff = 22 - 20 = 2
    uint256 availableReleases = timeAfterCliff / vesting.interval + 1;
```

```
    // uint256 availableReleases = 2 / 2 + 1 = 2

    return percentage + (intervalPercent * availableReleases);
    // return 40 + (10 + 2) = 60%
}
```

DFM-3 «Insufficient checks when adding addresses»

Status: Resolved

Severity: Low Risk

Description: The addWalletAddress function takes two arrays as arguments, which assume they are the same length, but this check is not performed.

Recommendation: To avoid errors due to invalid arguments being passed, it's best to process the arguments in the right way, for example:

```
function addWalletAddress(
    uint256 _vid,
    address[] memory _addresses,
    uint256[] memory _amounts
) internal onlyOwner {
    uint256 l = _addresses.length;
    require(l > 0, "Vesting: Arrays length should be > 0");
    require(l == _amounts.length, "Vesting: Incorrect arrays length");

    Vesting storage vesting = vestingInfo[_vid];
    for (uint256 i; i < l; ) {
        User storage user = users[_vid][_addresses[i]];
        if (user.amount == 0) {
            addresses[_vid].push(_addresses[i]);
        }
        user.walletAddress = _addresses[i];
        user.amount += _amounts[i];
        vesting.totalAmount += _amounts[i];
        unchecked { ++i; }
    }
    vesting.totalWallets = addresses[_vid].length;
}
```

DFM-4 «Insufficient checks when adding vesting»

Status: Resolved

Severity: Low Risk

Description: The addVesting function takes many arguments, which require additional checks to avoid errors and make the algorithms work correctly. In particular, the contract uses two types of vesting, for which the format of the arguments is different, which also requires additional checks.

Recommendation: Add additional checks to comply with development best practices. For example:

```
emun VestingType { Linear, Custom }

struct Vesting {
    // ..
    VestingType typ;
    // ..
}

function addVesting(
    VestingType _typ,
    string memory _name,
    address _tokenAddress,
    uint256 _interval,
    uint256[] memory _unlocks,
    uint256[] memory _percentages,
    address[] memory _addresses,
    uint256[] memory _amounts
) external onlyOwner {
    require(uint8(_typ) < 2, "Vesting: Incorrect type");
    require(_tokenAddress != address(0), "Vesting: Incorrect token address");

    if (_typ == VestingType.Linear) {
        require(_interval > 0, "Vesting: Interval should be > 0");
        require(_unlocks.length == 3, "Vesting: Unlocks length should be equal
3");
        require(_percentages.length == 1, "Vesting: Percentages length should be
equal 1");
    } else {
        require(_unlocks.length > 0, "Vesting: Arrays length should be > 0");
        require(_percentages.length == _unlocks.length >, "Vesting: Incorrect
arrays length");
    }

    uint256 vid = vestingInfo.length;

    vestingInfo.push(
        Vesting({
            id: vid,
            typ: _typ,
            name: _name,
            tokenAddress: _tokenAddress,
            interval: _interval,
            unlocks: _unlocks,
            percentages: _percentages,
            totalWallets: 0,
            totalAmount: 0,
            totalClaimed: 0
        })
    );

    addWalletAddress(vid, _addresses, _amounts);
```

```
}
```

Also, it's better practice to use a non-string value for the vid variable so that you can more easily interact with the variable, for example when comparing.Our example uses enum. Instead of comparing via keccak256(abi.encodePacked()), you can use _typ == VestingType.Linear or _typ == VestingType.Custom.

Also, checking the _addresses and _amounts arguments in the addVesting function is not required because they will be done in the addWalletAddress function.

DFM-5 «Potential loss of owner»

Status: Resolved

Severity: Low Risk

Description: There is no need to add the transferOwner function as OpenZeppelin's Ownable contract already contains it. In addition, your implementation uses the _transferOwnership function call instead of transferOwnership , which ignores the check for address(0), which can lead to accidental loss of access rights to the contract.

The Ownable contract also contains the renounceOwnership function, which resets the owner of the contract.

Recommendation: Most of the functions in your contract require remote permissions, and as a result, loss of permissions can become critical. The best solution would be to stop using your own transferOwner function and OpenZeppelin's own renounceOwnership function. For example, like this:

```
function renounceOwnership() public override onlyOwner {
    revert("Vesting: Renounce ownership disabled");
}
```

DFM-6 «Potentially incorrect late vesting logic»

Status: Open

Severity: Informational

Description: New addresses can be added to the vesting at any stage of the vesting, even if the vesting has already been completed - in this case, users will immediately be able to unlock 100% of the tokens.

In addition, when re-vesting for the same address, he will also be able to collect part of the funds for the intervals that have already passed.

Recommendation: Make sure that the logic matches your idea or fix the logic for adding addresses to the vesting. For example, you can add addresses to the vesting until the first unlock time has come or the cliff has ended (for linear vesting).

DFM-7 «Potentially reentrancy»

Status: Resolved

Severity: Informational

Description: Before interacting with third-party contracts and sending funds, it is always worth changing the contract storage first. In your case, the claim function first send tokens to the user, and then change the user's storage.

If you use only verified or native tokens, then there will be no vulnerability, but otherwise a reenterancy attack can be performed.

Recommendation: Make sure you only use your own or trusted tokens, or change the claim implementation to be more secure, like this:

```
function claim(uint256 _vid) public returns (bool) {
    require(_vid < vestingInfo.length, "Vesting: Incorrect vesting id");

    Vesting storage vesting = vestingInfo[_vid];
    User storage user = users[_vid][msg.sender];
    uint256 claimable = claimableTokens(_vid);

    require(claimable > 0, "Vesting: No tokens are due");

    user.claimed += claimable;
    vesting.totalClaimed += claimable;

    safeTOKENTransfer(_vid, msg.sender, claimable);
    return true;
}
```

DFM-8 «Using safeTransfer»

Status: Resolved

Severity: Informational

Description: To implement a safe transfer, it is better to use the SafeERC20 library from OpenZeppelin.

Recommendation: If you want to use a safer transfer implementation, you can change the code like this:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

using SafeERC20 for IERC20;

function safeTOKENTransfer(
    uint256 _id,
    address _to,
    uint256 _amount
) internal {
    IERC20 token = IERC20(vestingInfo[_id].tokenAddress);
    uint256 bal = token.balanceOf(address(this));
    require(bal >= _amount, "Vesting: Not enough tokens in treasury");

    if (_amount > 0) {
        token.safeTransfer(_to, _amount);
    }
}
```

DFM-9 «Outdated and unused libraries»

Status: Resolved

Severity: Informational

Description: The contract uses the SafeMath library, which is not relevant for Solidity>=0.8.0 (your version of Solidity is 0.8.8) - SafeMath is built into the Solidity compiler by default.

In addition, the contract imports "@openzeppelin/contracts/interfaces/IERC721Enumerable.sol", "@openzeppelin/contracts/utils/Strings.sol" and inherits "InterestHelper.sol" which are not used.

Recommendation: Stop using SafeMath and switch to classic arithmetic operations, as well as remove unnecessary imports.

DFM-10 «More secure calculations»

Status: Resolved

Severity: Informational

Description: Since there are no checks on input arguments when adding a vesting, there is no guarantee that the total percentage for the user will be 100%, which can lead to draining the token reserves in case of errors.

Recommendation: To provide at least minimal protection against various errors, you can add an additional check to the claimableTokens function, which will not allow you to receive more than 100% of the funds. For example, like this:

```
function claimableTokens(uint256 _vid) public view returns (uint256) {
    // ...

    uint256 userAmount = users[_vid][msg.sender].amount;
    claimable = claimable * userAmount / 1e20;
    if (claimable > userAmount) {
        claimable = userAmount;
    }
    return (claimable - users[_vid][msg.sender].claimed);
}
```

Automated Analyses

Slither

Slither's automatic analysis not found vulnerabilities, or these false positives results .

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, review open issue tickets, and investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system to make a final decision.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed