HYDN

Smart Contract Security Assessment

February 19, 2024

Project Name:

My Lovely Planet

Prepared by:

HYDN

Executive Summary

Client Name: My Lovely Planet

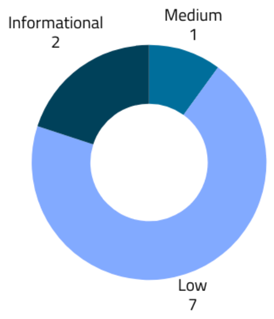
Language: Solidity

Timeline: Delivered 19 February 2024

Repository: https://github.com/My-Lovely-Planet/MLC-contracts

Method: Static Automated Analysis + Manual Review

Vulnerability Summary



10
Total Findings

9
Resolved

1
Acknowledged

Contents

Introduction

About HYDN

HYDN is an industry leader in blockchain security and smart contract audits. Founded by Warren Mercer, a world renowned cybersecurity and blockchain expert, who has previously held senior roles at NYSE, Cisco, and Alert Logic. Having been involved in cryptocurrency for over 10 years, Warren is dedicated to making the blockchain ecosystem as secure as it can be for everyone. Warren serves as the CEO for HYDN and heads up the delivery team to ensure that work is carried out to the highest standard.

The HYDN delivery team has over 10 years combined experience in blockchain, having performed smart contract audits and built security systems for a large range of protocols and companies. HYDN have performed smart contract security services for the likes of SushiSwap, Bittrex Global, Sablier, Telos, My Lovely Planet , Swapsicle, Dancing Seahorse, Nau , CrossWallet, BlueZilla, and many more.

HYDN worked closely with My Lovely Planet to consider their unique business needs, objectives, and concerns. Our mission is to ensure the blockchain supports secure business operations for all and he has built the team at HYDN from the ground up to meet this very personal objective.

To keep up to date with our latest news and announcements, check out our website https://hydnsec.com/ or follow @hydnsecurity on Twitter.

About My Lovely Planet

My Lovely Planet is a Play to Save game. The main concept is that everything you do in the game has a positive impact in the real world. The My Lovely Coin is an ERC20 utility token that is based on the Polygon blockchain.

Methodology

When tasked with conducting a security assessment, HYDN works through multiple phases of security auditing to ensure smart contracts are audited thoroughly. To begin the process automated tests are carried out, before HYDN then moves onto carrying out a detailed manual review of the smart contracts.

HYDN uses a variety of open-source tools and analyzers as and when they are required and alongside this, HYDN primarily focuses on the following classes of security and reliability issues:

Basic Coding Mistakes

One of the most common causes of critical vulnerabilities is basic coding mistakes. Countless projects have experienced hacks and exploits due to simple, surface level mistakes that could have been flagged and addressed by a simple code review. The HYDN automated audit process which includes model checkers, fuzzers, and theorem provers analyses the smart contract for many of these basic coding mistakes. Once the automated audit has taken place, HYDN then performs a manual review of the code to gain familiarity with the contracts.

Business Logic Risks

HYDN reviews the platform or projects design documents before analysing the code to ensure that the team has a deep understanding of the business logic and goals. Following this, HYDN reviews the smart contracts to ensure that the contract logic is in line with the expected functionality.

HYDN also analyses the code for inconsistencies, flaws, vulnerabilities, or non-technical business risks which could impact business logic such as tokenomics errors, arbitrage opportunities, and share pricing.

Complex Integration Risks

Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem.
We perform a meticulous review of all of the contract's possible external interactions, and summarise the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code Maturity

HYDN reviews the smart contracts for potential improvements in the codebase. These improvements ensure that the code follows industry best practices and guidelines, or code quality standards. Alongside this, HYDN makes suggestions for code optimization items such as gas optimization, upgradeability weaknesses, centralization risks, and more.

Impact Rankings

Once HYDN has completed the security assessment, each issue is assigned an impact rating. This impact rating is based upon both the severity and likelihood of the issue occurring. Each security assessment is different and the business needs and goals, such as project timelines, and My Lovely Planet threat modelling. are taken into account when the impact rankings are created.

HYDN assigns the following impact rankings: Critical, High, Medium, Low (listed by severity).

Disclaimer

This HYDN security assessment does not provide any warranties on finding all possible issues within the scope and the evaluation results do not guarantee the absence of any issues. **HYDN cannot make any guarantees on any further code which is added or altered after the security assessment has taken place**. As a single security assessment can never be considered fully comprehensive, HYDN always recommends multiple independent assessments paired with a bug bounty program. This security assessment report should not be considered as financial or investment advice.

Summary of Findings

Executive Summary

From January 15th through to January 22nd 2024, HYDN was tasked with performing a Smart Contract Audit on the My Lovely Planet Smart Contracts. Initially, the assessment focused on gaining a deep understanding of the contracts and My Lovely Planet business logic, as well as drilling down into core functions. Our team then spent the remaining time manually auditing the codebase and ideating potential attack vectors.

During the assessment, the HYDN audit team found a number of issues, including One Medium issue and Several further Low and Informational issues.

Project Summary

Delivery Date	19 February 2024
Language	Solidity
Repository	https://github.com/My-Lovely-Planet/MLC-contracts
Scope	https://github.com/My-Lovely-Planet/MLC-contracts/blob/main/contracts/MyLovelyCoin.sol https://github.com/My-Lovely-Planet/MLC-contracts/blob/main/contracts/Vesting.sol

	https://github.com/My-Lovely-Planet/MLC-contracts/blob/main/contracts/VestingAggregator.sol
Initial Commits	448da68ff93929f9c20532343aea2b7fc95c351c
Latest Commit	b5ffaab7e5875d5eaa9d67ab63574cc942950c62

Breakdown of Findings

ID	Title	Severity	Status
1	Vesting Update Allocation Open To Abuse By Transaction Ordering	Medium	Resolved
2	Start Vesting At Deterministic Date	Low	Resolved
3	Payable Constructor Without Withdraw Function	Low	Resolved
4	Remove Redundant Storage Variable	Low	Resolved
5	Gas optimization - Unnecessary Non Re-Entrance Mutex Lock Protection	Low	Resolved
6	Gas Optimization - Unnecessary Safe Transfer	Low	Resolved

7	Gas optimization - Flag Function As External When Not Used Internally	Low	Resolved
8	Use The Latest Solidity Compiler Version	Low	Resolved
9	Emit Event On Important Storage state Changes	Informational	Resolved
10	Centralization Risks	Informational	Acknowledged

Detailed Findings

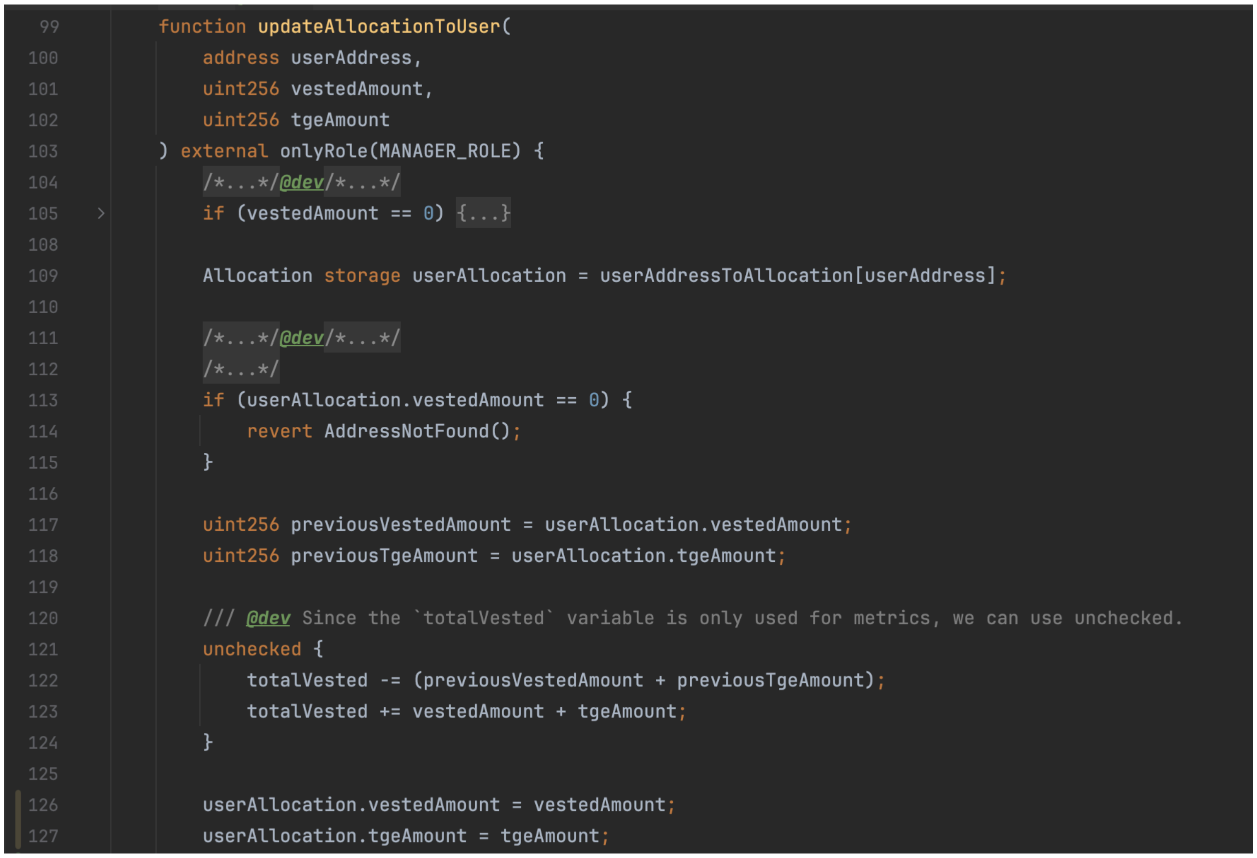
1 Vesting Update Allocation Open To Abuse By Transaction Ordering [Medium]

1.1 Description

In the Vesting.sol contract, the update allocation function intends to modify the existing one-user allocation after it has been set.

Scenario -
If we consider a case in which a malicious user with tgeAmount not null set at the first initialization, and not claimed, the following issue could occur.

1) A Manager who has been granted role access broadcasts a transaction to update vestedAmount and/or tgeAmount
2) A Malicious User watching for such transaction will be able to front-run the Manager transaction and claim the initially defined tgeAmount + (partial/full) vestedAmount.
3) After this, the Manager transaction will override the now zero (or remaining vestedAmount) + tgeAmount allocation.



```
function updateAllocationToUser(
    address userAddress,
    uint256 vestedAmount,
    uint256 tgeAmount
) external onlyRole(MANAGER_ROLE) {
    /*...*/@dev/*...*/
    if (vestedAmount == 0) {...}

    Allocation storage userAllocation = userAddressToAllocation[userAddress];

    /*...*/@dev/*...*/
    /*...*/
    if (userAllocation.vestedAmount == 0) {
        revert AddressNotFound();
    }

    uint256 previousVestedAmount = userAllocation.vestedAmount;
    uint256 previousTgeAmount = userAllocation.tgeAmount;

    /// @dev Since the `totalVested` variable is only used for metrics, we can use unchecked.
    unchecked {
        totalVested -= (previousVestedAmount + previousTgeAmount);
        totalVested += vestedAmount + tgeAmount;
    }

    userAllocation.vestedAmount = vestedAmount;
    userAllocation.tgeAmount = tgeAmount;
```

Extract of Vesting.sol

1.2 Impact

In the context of the scenario detailed above, this will allow a malicious user to claim the initial allocation upfront for the update and, in addition, claim the upcoming new allocation.

1.3 Recommendations

Regarding the immutable business requirements for the distribution, it makes sense to consider the initial allocation to be accurate, so it may be acceptable to fully remove the update logic.

According to the comments this function is only here in case of a mistake during the allocation set phase, so it is not strictly needed as including this may open a new attack vector.

The manager can withdraw all funds in case of unexpected issues with the contract and this will act as the ultimate edge and cover the needs of the update allocation function.

Another alternative approach would be to freeze user allocations to *'lock'* in the current state (*tgeAmount is claimed? How much vestedAmount remains?*) and then safely update to a new state ensuring the state transition correctness.

1.4 Remediation

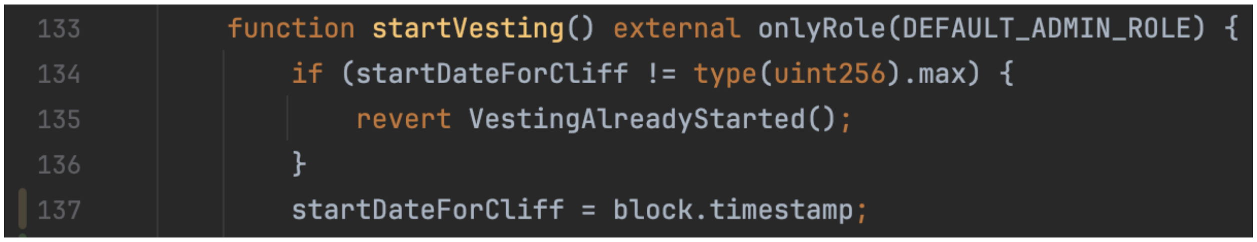
Fixed in the latest commit.

2 Start Vesting At Deterministic Date [Low]

2.1 Description

Multiple vesting contracts will be deployed, according to multiple vesting durations. The manager will start the vesting process by calling startVesting() function. This will allow the tge amount to be claimed, and start the Cliff and Vesting Period.

The start date used is the block.timestamp, so this is the exact time when the transaction will be minted. The core issue here is that Block Timestamp can't be predicted in a deterministic way.



```
    function startVesting() external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (startDateForCliff != type(uint256).max) {
            revert VestingAlreadyStarted();
        }
        startDateForCliff = block.timestamp;
```

Extract of Vesting.sol

2.2 Impact

From a business requirements perspective, the legitimate expectation is fairness between all vesting participants, so all vesting must start at the same time to ensure fair launch distribution.

As Block Timestamp can't be predicted exactly, ensuring all vesting calls to startVesting() to be minted within the block will be extremely difficult or impossible.

This will result in an unfair advantage for the first vesting contract started, as tgeAmount will be available and the Cliff end will be reached before others.

2.3 Recommendations

In general do not rely on block.timestamp. In this specific situation adding a parameter to startVesting to set a precise timestamp in place of using the block.timestamp will allow the manager to start all vesting at the same time.

Ensure the new startDateForCliff setter parameter is not a past timestamp and is far enough in the future to allow the manager to broadcast and mint this transaction for all the vesting contracts.

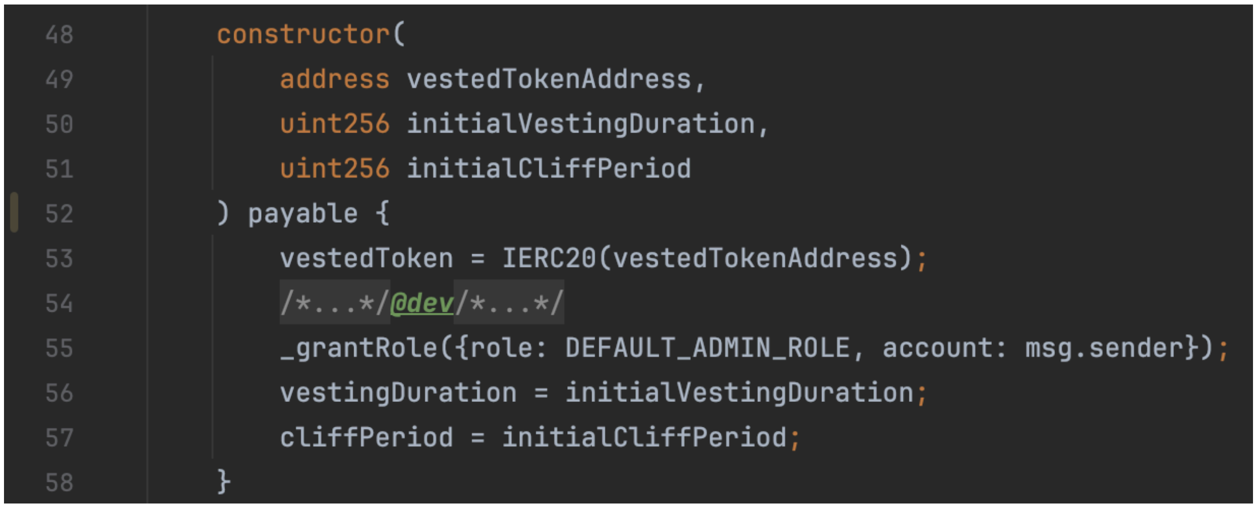
2.4 Remediation

Fixed in the latest commit. The startVesting() function allows setting a timestamp that activates vesting. This method can only be activated once.

3 Payable Constructor Without Withdraw Function [Low]

3.1 Description

The vesting contract constructor is flagged as Payable, but according to the contract logic, it isn't supposed to hold native tokens at any time. In addition, no withdrawal function exists.



```
constructor(
    address vestedTokenAddress,
    uint256 initialVestingDuration,
    uint256 initialCliffPeriod
) payable {
    vestedToken = IERC20(vestedTokenAddress);
    /*...*/@dev/*...*/
    _grantRole({role: DEFAULT_ADMIN_ROLE, account: msg.sender});
    vestingDuration = initialVestingDuration;
    cliffPeriod = initialCliffPeriod;
}
```

Extract of Vesting.sol

3.2 Impact

The impact of this is that any native token sent during the contract deployment process will be lost.

3.3 Recommendations

According to the business logic, the contract constructor is not supposed to receive any native tokens. This can be seen as a gas optimization strategy as the compiler will not check the sent amount during contract initialization, but it opens a risk of loss of funds during deployment.

This contract function, loop, and storage are far from being optimized and there is a lot of unnecessary read logic, so depending on the purpose of this Payable logic, it can be kept as it is or removed altogether.

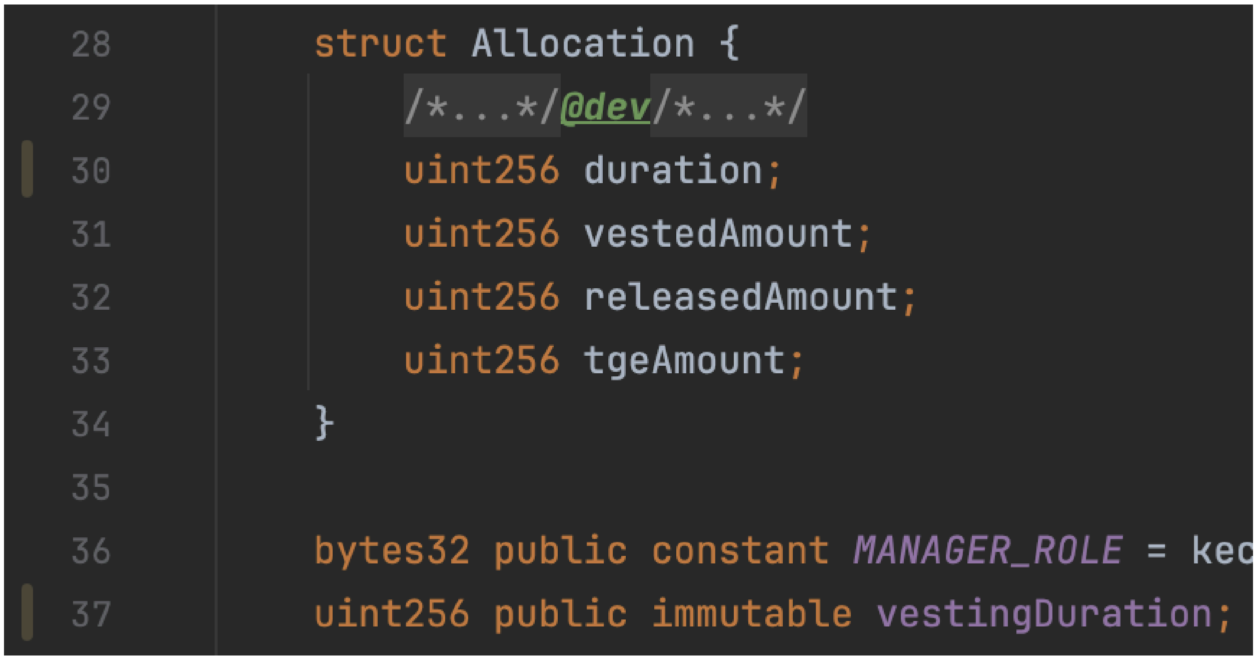
3.4 Remediation

Fixed in the latest commit. The contract is no longer payable, and it is not possible to send native tokens to the contract unless forced through self-destruct- but there is no control over that.

4 Remove Redundant Storage Variable [Low]

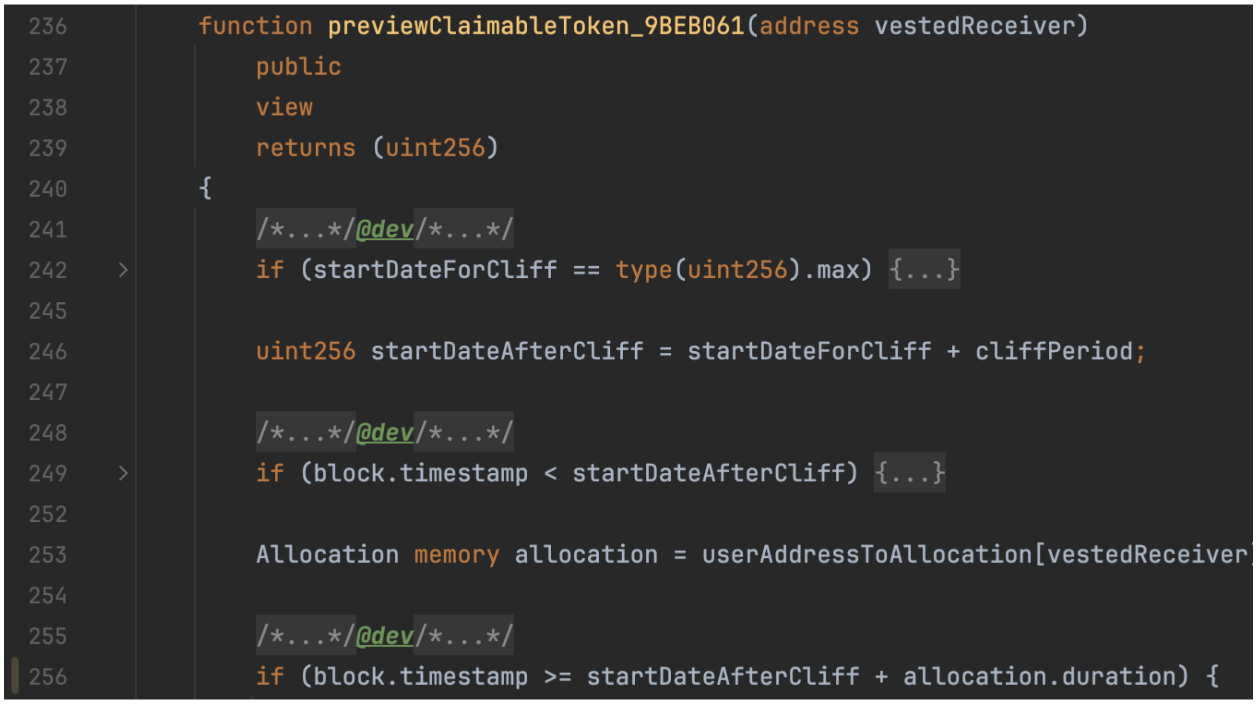
4.1 Description

Multiple vesting contracts will be deployed with different durations. All users with the allocation vesting duration claim will be held under the same contract, one vesting period = one vesting contract. The allocation structure, used to store peer user allocation, also contains the vesting duration. This is a duplicate from the top-level contract immutable variable vestingDuration.



```
    struct Allocation {
        /*...*/@dev/*...*/
        uint256 duration;
        uint256 vestedAmount;
        uint256 releasedAmount;
        uint256 tgeAmount;
    }

    bytes32 public constant MANAGER_ROLE = kec
    uint256 public immutable vestingDuration;
```

Extract of Vesting.sol



```
    function previewClaimableToken_9BEB061(address vestedReceiver)
        public
        view
        returns (uint256)
    {
        /*...*/@dev/*...*/
        if (startDateForCliff == type(uint256).max) {...}

        uint256 startDateAfterCliff = startDateForCliff + cliffPeriod;

        /*...*/@dev/*...*/
        if (block.timestamp < startDateAfterCliff) {...}

        Allocation memory allocation = userAddressToAllocation[vestedReceiver

        /*...*/@dev/*...*/
        if (block.timestamp >= startDateAfterCliff + allocation.duration) {
```

Extract of Vesting.sol

4.2 Impact

Including this redundant storage variable will increase the deployment costs, admin allocation transactions costs, and user claim transaction costs.

4.3 Recommendations

HYDN recommends removing the duration field from the Allocation structure.

4.4 Remediation

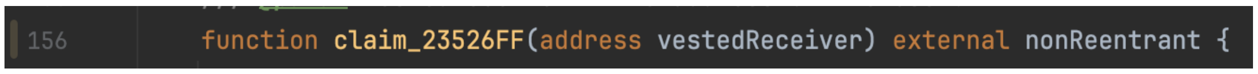
Fixed in the latest commit. The Allocation structure has been updated.

5 Gas optimization - Unnecessary Non Re-Entrance Mutex Lock Protection [Low]

5.1 Description

The claim function uses a nonReentrant modifier to protect against re-entrancy attacks. In this case, there are no external dependencies and one external call that is the transfer of MLC tokens.

As this token is also part of and under the control of My Lovely Planet, it is a minimal and fully compliant ERC-20 token inherited from the OpenZeppelin official implementation, so therefore no hook and intermediary step can result in a re-entrance attack.



```
function claim_23526FF(address vestedReceiver) external nonReentrant {
```

Extract of Vesting.sol

5.2 Impact

Here the unnecessary memory and storage read/write will increase the gas costs at contract deployment time and execution time- in this case when the user will call the claim function.

5.3 Recommendations

HYDN recommends removing unnecessary nonReentrant modifiers.

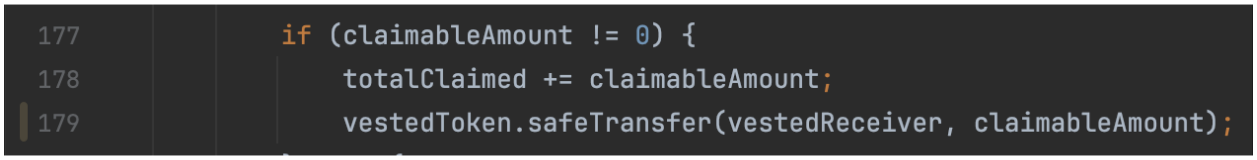
5.4 Remediation

Fixed in the latest commit.

6 Gas Optimization - Unnecessary Safe Transfer [Low]

6.1 Description

The claim function computes the available amount for the user at a specific point in time to withdraw. The underlying token is My Lovely Coin- it is a minimal and fully compliant ERC-20 token inherited from OpenZeppelin official implementation so the transfer function does not need to be wrapped into a utility function safeTransfer to ensure optimal return, as MLC is the only token that will be used.



```
    if (claimableAmount != 0) {
        totalClaimed += claimableAmount;
        vestedToken.safeTransfer(vestedReceiver, claimableAmount);
```

.Extract of Vesting.sol

6.2 Impact

Unnecessary logic will increase the gas cost at contract deployment time and at execution time- in this case when the user will call the claim function.

6.3 Recommendations

HYDN recommends removing the unnecessary SafeERC20 library and by consequence the safeTransfer wrap function.

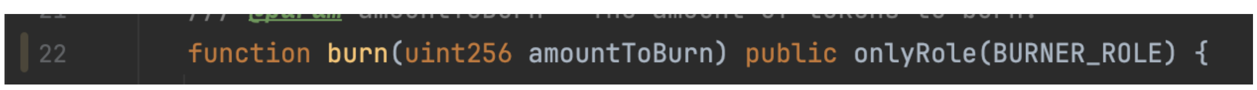
6.4 Remediation

Fixed in the latest commit.

7 Gas optimization - Flag Function As External When Not Used Internally [Low]

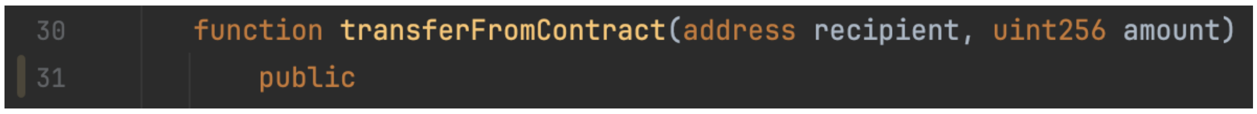
7.1 Description

Functions that are called exclusively by a third-party contract or EOA must be flagged as external.



```
function burn(uint256 amountToBurn) public onlyRole(BURNER_ROLE) {
```

Extract of MyLovelyCoin.sol



```
function transferFromContract(address recipient, uint256 amount)
    public
```

Extract of MyLovelyCoin.sol

7.2 Impact

Flagging functions as Public instead of External will increase contract deployment costs and execution call costs.

7.3 Recommendations

Change burn() and transferFromContract() visibility from public to external.

7.4 Remediation

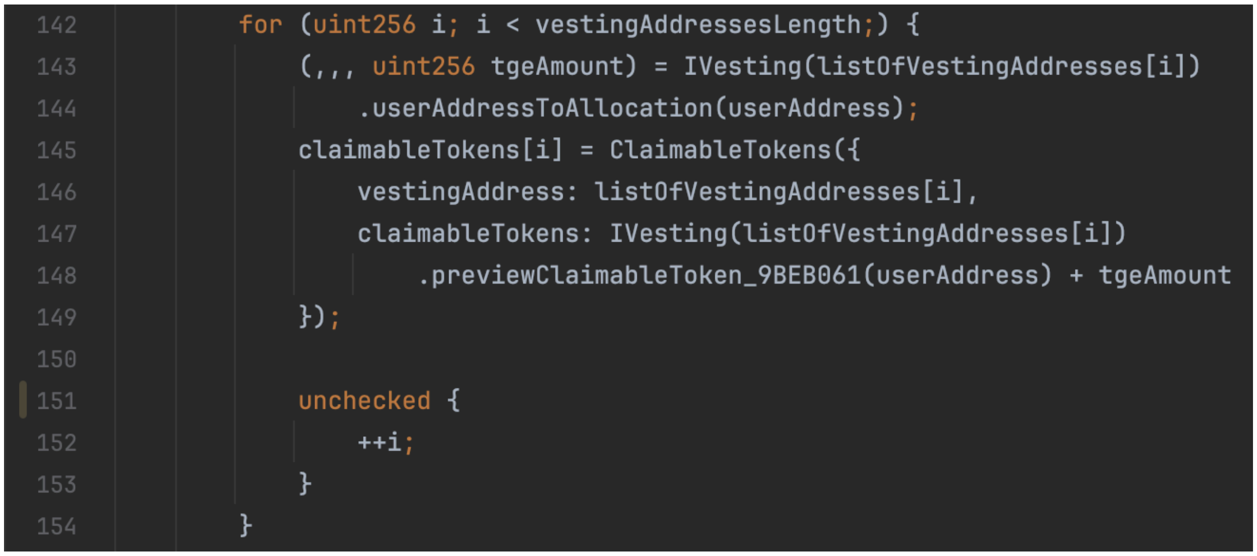
Fixed in the latest commit.

8 Use The Latest Solidity Compiler Version [Low]

8.1 Description

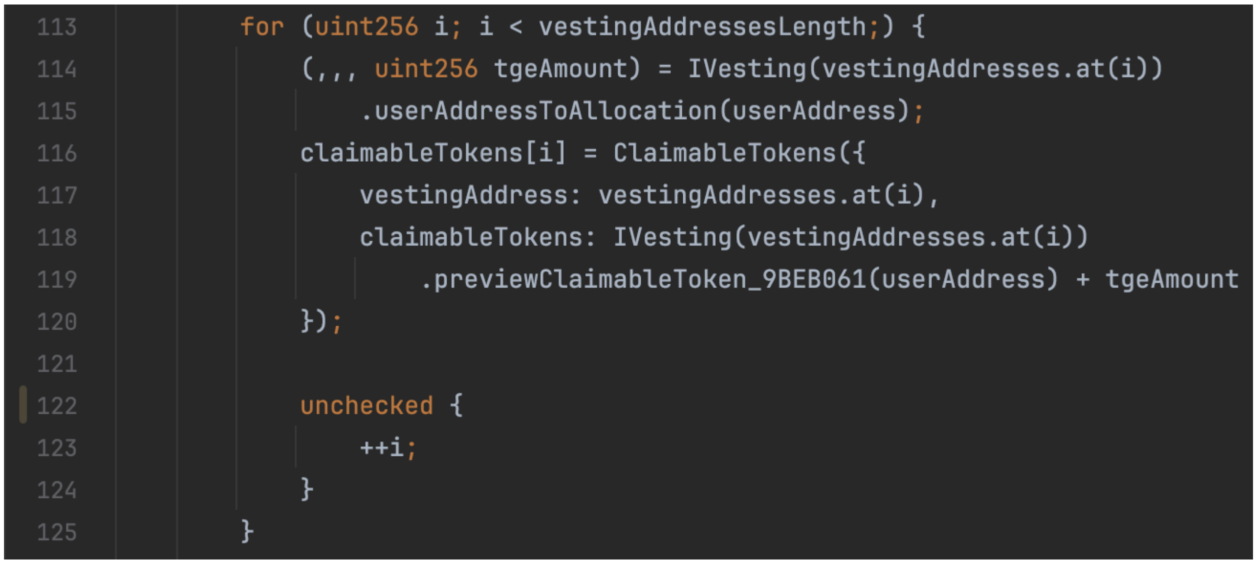
HYDN recommends always using the latest version of the Solidity compiler at deployment time to take advantage of beneficial bug and vulnerability patches and compiler improvement.

Gas optimization for loop iterators under uncheck blocks is not necessary as this optimization is already handled natively by the latest version of the compiler and improves readability.



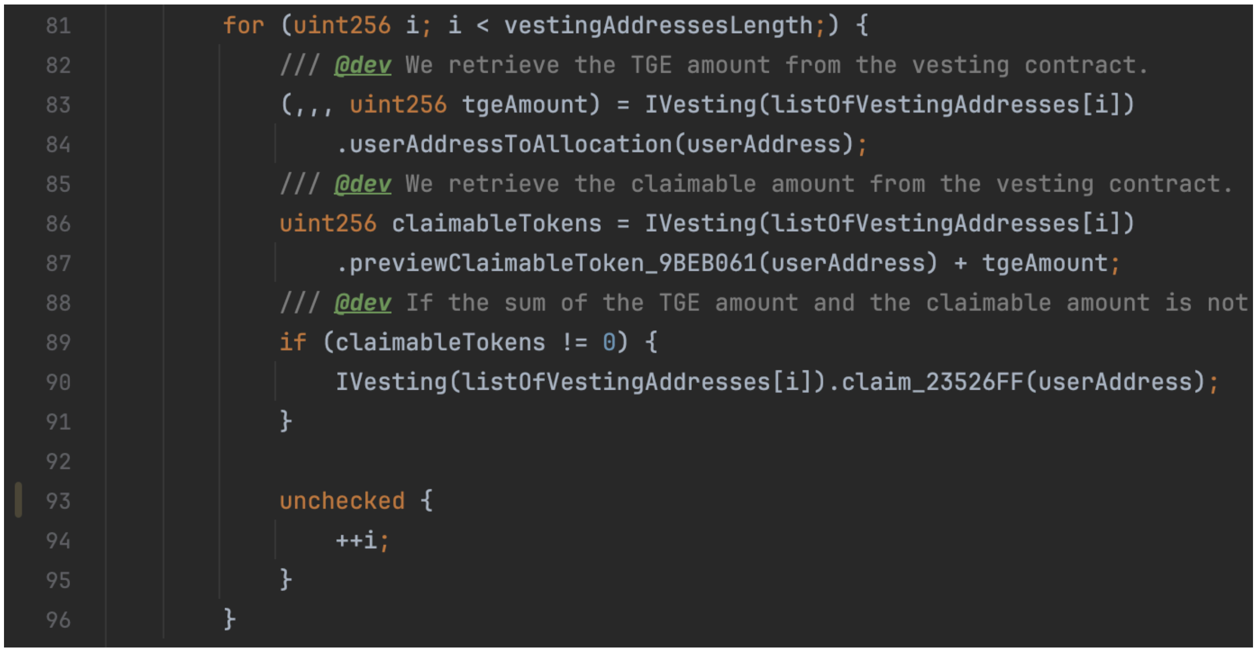
```
for (uint256 i; i < vestingAddressesLength;) {
    (,,, uint256 tgeAmount) = IVesting(listOfVestingAddresses[i])
        .userAddressToAllocation(userAddress);
    claimableTokens[i] = ClaimableTokens({
        vestingAddress: listOfVestingAddresses[i],
        claimableTokens: IVesting(listOfVestingAddresses[i])
            .previewClaimableToken_9BEB061(userAddress) + tgeAmount
    });

    unchecked {
        ++i;
    }
}
```

Extract of VestingAggregator.sol



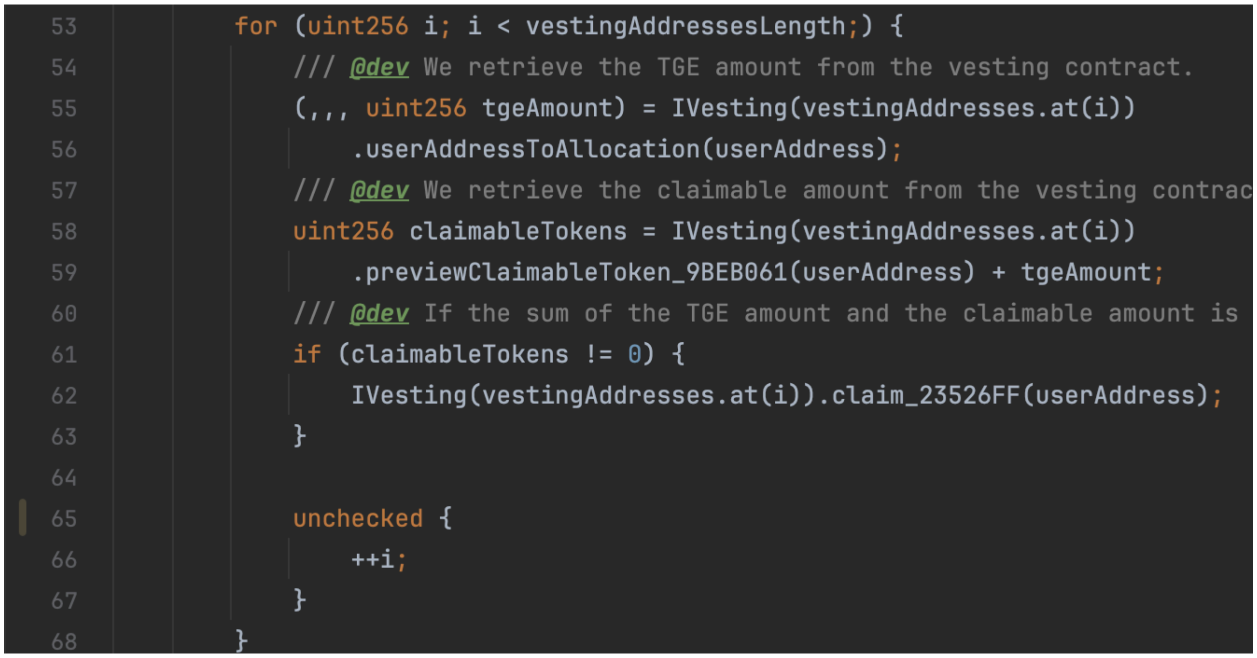
```
    for (uint256 i; i < vestingAddressesLength;) {
        (,,, uint256 tgeAmount) = IVesting(vestingAddresses.at(i))
            .userAddressToAllocation(userAddress);
        claimableTokens[i] = ClaimableTokens({
            vestingAddress: vestingAddresses.at(i),
            claimableTokens: IVesting(vestingAddresses.at(i))
                .previewClaimableToken_9BEB061(userAddress) + tgeAmount
        });

        unchecked {
            ++i;
        }
    }
```

Extract of VestingAggregator.sol



```
    for (uint256 i; i < vestingAddressesLength;) {
        /// @dev We retrieve the TGE amount from the vesting contract.
        (,,, uint256 tgeAmount) = IVesting(listOfVestingAddresses[i])
            .userAddressToAllocation(userAddress);
        /// @dev We retrieve the claimable amount from the vesting contract.
        uint256 claimableTokens = IVesting(listOfVestingAddresses[i])
            .previewClaimableToken_9BEB061(userAddress) + tgeAmount;
        /// @dev If the sum of the TGE amount and the claimable amount is not
        if (claimableTokens != 0) {
            IVesting(listOfVestingAddresses[i]).claim_23526FF(userAddress);
        }

        unchecked {
            ++i;
        }
    }
```

Extract of VestingAggregator.sol



```
        for (uint256 i; i < vestingAddressesLength;) {
            /// @dev We retrieve the TGE amount from the vesting contract.
            (,,, uint256 tgeAmount) = IVesting(vestingAddresses.at(i))
                .userAddressToAllocation(userAddress);
            /// @dev We retrieve the claimable amount from the vesting contrac
            uint256 claimableTokens = IVesting(vestingAddresses.at(i))
                .previewClaimableToken_9BEB061(userAddress) + tgeAmount;
            /// @dev If the sum of the TGE amount and the claimable amount is
            if (claimableTokens != 0) {
                IVesting(vestingAddresses.at(i)).claim_23526FF(userAddress);
            }

            unchecked {
                ++i;
            }
        }
```

Extract of VestingAggregator.sol

8.2 Impact

Using outdated versions of the Solidity compiler can pose several risks and dangers, particularly in the context of development and deployment. In a similar way to relying on outdated third party dependencies, using older versions of the Solidity compiler increases the risk of being exposed to latent or future vulnerabilities that may have been fixed in later versions.

Another key reason to ensure you are using the latest version is because newer versions often introduce new features, optimizations, and syntactical improvements. Using an outdated version may prevent developers from leveraging these advancements, leading to less efficient and potentially less secure code.

Newer compiler versions also often include optimizations that improve the performance of the compiled bytecode. Using older versions may result in slower and more costly (in terms of gas) contract execution.

8.3 Recommendations

- Use the latest Solidity compiler 0.8.23 in place of 0.8.4.
- Use the latest OpenZeppelin library version 4.9.5 in place of 4.9.2.

8.4 Remediation

Fixed in the latest commit, although using the Paris flag for EVM as there are doubts about the support of Shanghai on Polygon Mainnet.

9 Emit Event On Important Storage State Change [Low]

9.1 Description

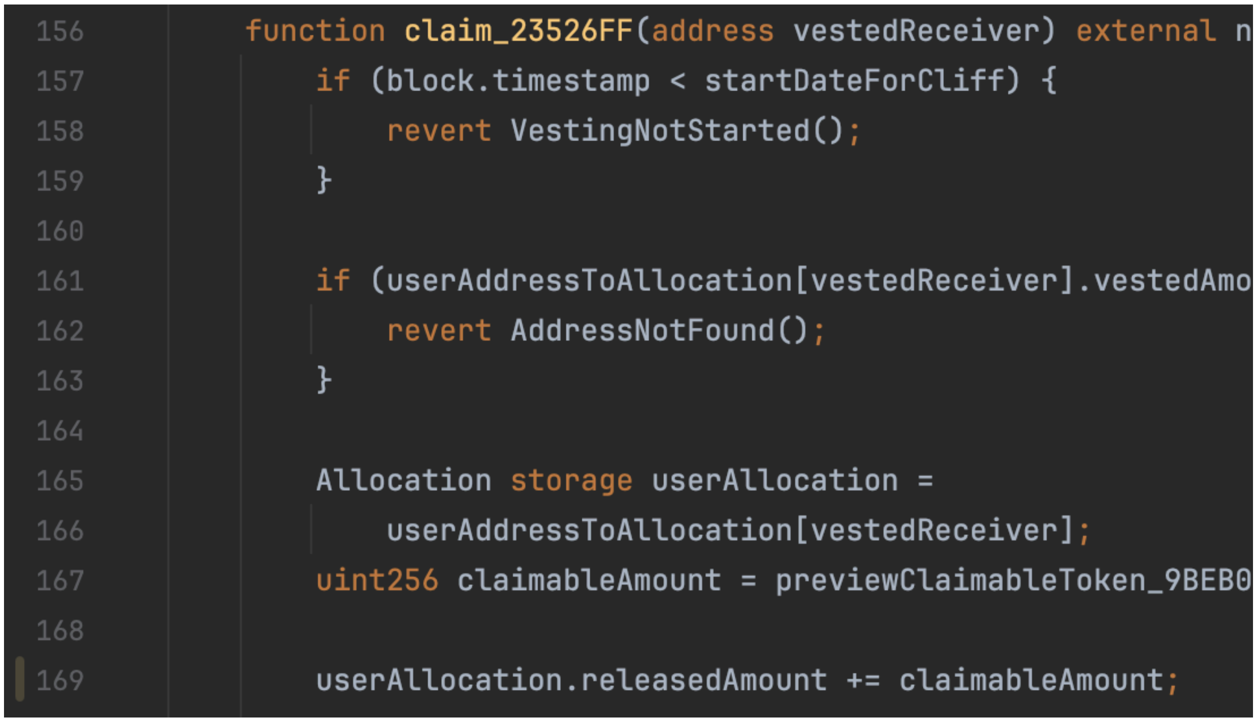
Emitting events on important actions such storage changes during a contract function call is recommended as best practice for retrieval of what is going on and simple data read.

See:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/finance/VestingWallet.sol#L31-L32

9.2 Recommendations

Emit event at:

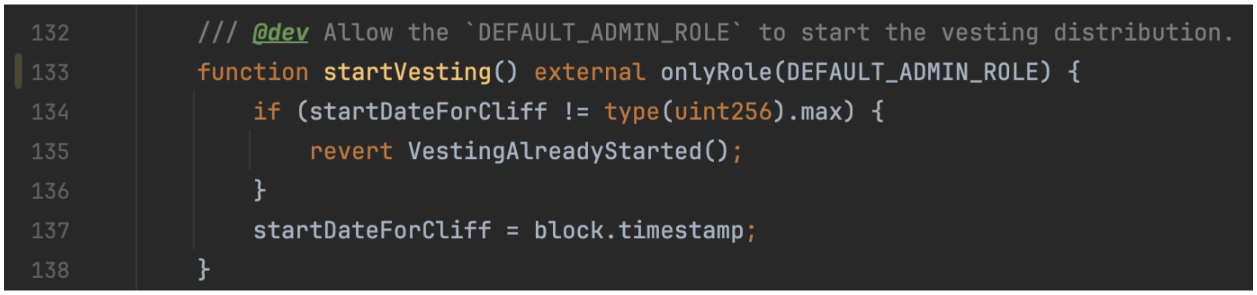
User claim, releasedAmount updated:



```
    function claim_23526FF(address vestedReceiver) external n
        if (block.timestamp < startDateForCliff) {
            revert VestingNotStarted();
        }

        if (userAddressToAllocation[vestedReceiver].vestedAmo
            revert AddressNotFound();
        }

        Allocation storage userAllocation =
            userAddressToAllocation[vestedReceiver];
        uint256 claimableAmount = previewClaimableToken_9BEB0

        userAllocation.releasedAmount += claimableAmount;
```

Extract of Vesting.sol

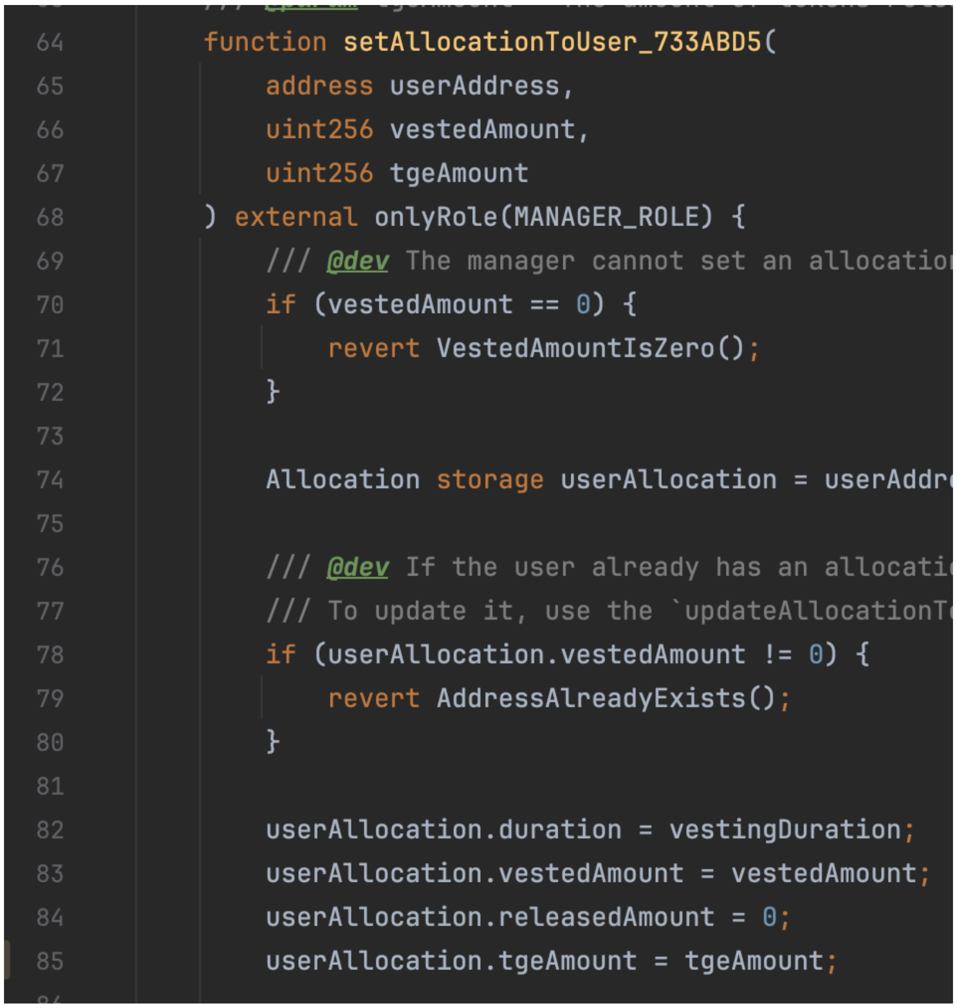
StartVesting():



```
    /// @dev Allow the `DEFAULT_ADMIN_ROLE` to start the vesting distribution.
    function startVesting() external onlyRole(DEFAULT_ADMIN_ROLE) {
        if (startDateForCliff != type(uint256).max) {
            revert VestingAlreadyStarted();
        }
        startDateForCliff = block.timestamp;
    }
```

Extract of Vesting.sol

SetAllocation(), this may be worth considering, but please note that this would also increase gas costs



```
    function setAllocationToUser_733ABD5(
        address userAddress,
        uint256 vestedAmount,
        uint256 tgeAmount
    ) external onlyRole(MANAGER_ROLE) {
        /// @dev The manager cannot set an allocatio
        if (vestedAmount == 0) {
            revert VestedAmountIsZero();
        }

        Allocation storage userAllocation = userAddr

        /// @dev If the user already has an allocati
        /// To update it, use the `updateAllocationT
        if (userAllocation.vestedAmount != 0) {
            revert AddressAlreadyExists();
        }

        userAllocation.duration = vestingDuration;
        userAllocation.vestedAmount = vestedAmount;
        userAllocation.releasedAmount = 0;
        userAllocation.tgeAmount = tgeAmount;
```

Extract of Vesting.sol

9.4 Remediation

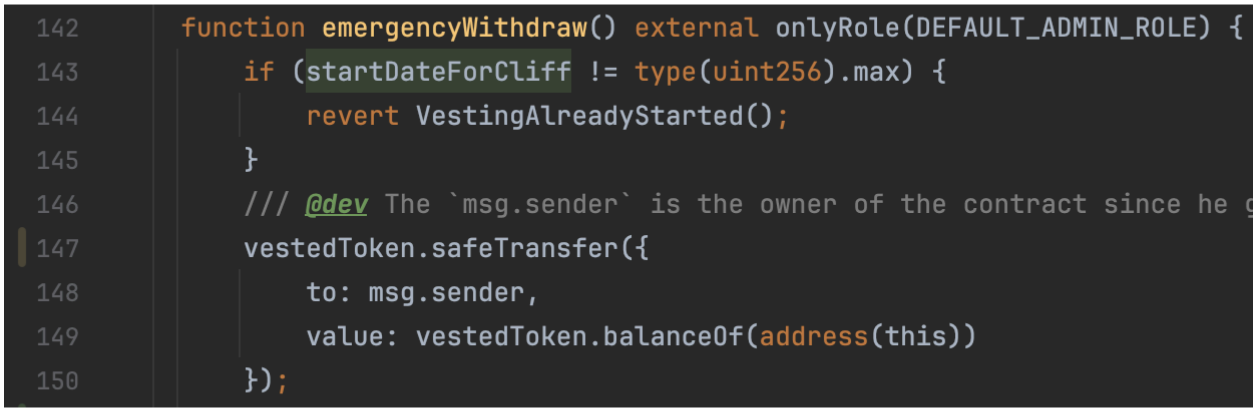
Fixed in the latest commit.

10 Centralization Risks [Informational]

10.1 Description

Due to the decentralized and immutable nature of the blockchain, centralization and restricted access control are risks for any user who interacts with the contract.

The vesting contract emergencyWithdraw() function allows the manager to withdraw to their own address at any time all remaining MLC tokens locked into the vesting contract.



```
function emergencyWithdraw() external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (startDateForCliff != type(uint256).max) {
        revert VestingAlreadyStarted();
    }
    /// @dev The `msg.sender` is the owner of the contract since he g
    vestedToken.safeTransfer({
        to: msg.sender,
        value: vestedToken.balanceOf(address(this))
    });
```

.*Extract of Vesting.sol*

10.2 Impact

Users must fully trust the contract owner to not perform this action maliciously, as if they did, it would result in the transfer of tokens to the contract owner.

10.3 Remediation

Acknowledged, but no action taken.