Kambria's Vesting Module

Smart Contract Security Audit

Prepared by ShellBoxes

Jan 12th, 2022 - Jan 13th, 2022

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kambria
Version	1.0
Classification	Public

Scope

Contract Name	Contract Address
VestingModule	0x3a26A3cB0f850070EB7bb97300A7adCD518E2CE5

Re-Audit

Contract Name	Contract Address
VestingModule	0xD973331c3Ae062070621A3D218dd5A8a4da08104

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kambria engaged ShellBoxes to conduct a security assessment on the Kambria's Vesting Module beginning on Jan 12th, 2022 and ending Jan 13th, 2022. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kambria

Kambria, an open innovation platform for Deep Tech.

Issuer	Kambria
Website	`https://kambria.io`
Type	Solidity Smart Contract
Documentation	Kambria DAO LP token Vesting contract document
udit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Kambria's Vesting Module implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 2 critical-severity, 1 high-severity, 1 medium-severity, 4 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. The user can lose his funds due to rounding errors	CRITICAL	Fixed
SHB.2. The contract is not guaranteed to be funded	CRITICAL	Fixed
SHB.3. The admin can withdraw the allocated amounts	HIGH	Fixed
SHB.4. The initialize function can be front-run	MEDIUM	Fixed
SHB.5. Owner can renounce ownership	LOW	Fixed
SHB.6. Unchecked arrays lengths	LOW	Fixed
SHB.7. Missing address and value verification	LOW	Fixed
SHB.8. Floating pragma	LOW	Fixed

3 Finding Details

SHB.1 The user can lose his funds due to rounding errors

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

The releasable function determines the amount that can be made available at a particular time, using the timestamp of the function call, the vesting duration, and the allocated amount and timestamp of the previous release function call. However, there is a chance of encountering rounding errors, which may result in a loss of funds for the user.

Exploit Scenario:

As an example, consider a vesting period of 1000 seconds and a user allocation of 100:

- The user calls the release function after 19 seconds from the start
- The expected releasable amount would be 19 * 100 / 1000 = 1.9

However, Solidity does not support floating points, so this value will round to one. This means that the lastClaimedTimestamp[claimer] timestamp will be updated with the new value of block.timestamp, resulting in a permanent loss of funds for the user, which is 0.9 in this case. This vulnerability can have a greater impact and result in the loss of substantial amounts from the user's end if the vesting term is longer.

Files Affected:

SHB.1.1: VestingModule.sol

```
function releasable(address claimer) public view returns (uint256) {
    //Before the vesting begins
    if (block.timestamp <= start) {
```

```
        return 0;
    }

    uint256 lastClaimedTimestamp_ = lastClaimedTimestamp[claimer];
    uint256 totalAllocation_ = totalAllocation[claimer];

    if (lastClaimedTimestamp_ == 0) {
        lastClaimedTimestamp_ = start;
    }

    // After the end of vesting
    if (block.timestamp >= start + duration) {
        return
            ((start + duration - lastClaimedTimestamp_) *
                totalAllocation_) / duration;
    }

    // During the vesting period
    return
        ((block.timestamp - lastClaimedTimestamp_) * totalAllocation_) /
        duration;
}
```

Recommendation:

The main issue is updating the lastClaimedTimestamp[claimer] to block.timestamp. After doing the math, we found that the lastClaimedTimestamp[claimer] should be updated using the following code:

- In the release function:

SHB.1.2: VestingModule.sol

```
uint256 timeElapsed = block.timestamp - lastClaimedTimestamp[msg.sender
    ↪ ];
uint256 divisionRest = timeElapsed * totalAllocation[msg.sender] %
    ↪ duration;
```

```
lastClaimedTimestamp[msg.sender] =lastClaimedTimestamp[msg.sender] +
    ↪ timeElapsed - divisionRest / totalAllocation[msg.sender];
```

- In the releaseTo function:

SHB.1.3: VestingModule.sol

```
uint256 timeElapsed = block.timestamp - lastClaimedTimestamp[receiver];
uint256 divisionRest = timeElapsed * totalAllocation[receiver] %
    ↪ duration;
lastClaimedTimestamp[receiver] =lastClaimedTimestamp[receiver] +
    ↪ timeElapsed - divisionRest / totalAllocation[receiver];
```

This implementation assures that the user does not lose any funds of his allocations independently of when he calls the release or the releaseTo functions.

Updates

The Kambria team resolved the issue by implementing the use of the recommended code.

SHB.2 The contract is not guaranteed to be funded

- Severity : CRITICAL
- Likelihood : 3
- Status : Fixed
- Impact : 3

Description:

The addAllocations function is used by the admin to allocate a specific amount to each one of the users. However, the contract is not guaranteed to be funded before allocation, which means the users who have allocated amounts in the contract might not be able to withdraw their releasable amounts.

Files Affected:

SHB.2.1: VestingModule.sol

```
function addAllocations (
    address[] memory addresses,
    uint256[] memory allocations
) external onlyRole(ADMIN_ROLE) {
    for (uint256 i = 0; i < addresses.length; i++) {
        // Check if added allocation
        //uint256 totalAllocation_ = totalAllocation(addresses[i]);
        require(totalAllocation[addresses[i]] == 0, "VestingModule: There
           ↪ are a addressed has been added allocation already");
    }
     for (uint256 i = 0; i < addresses.length; i++) {
        totalAllocation[addresses[i]] = allocations[i];

    }
```

Recommendation:

Consider approving the sum of the allocations array's elements prior to running addAllocations and adding a safeTransform call to the function to claim the approved funds. Thus, users will be able to withdraw their releasable funds at any time with a high guarantee.

Updates

The Kambria team resolved the issue by verifying the contract's balance to be sufficient to fulfill all the allocations.

SHB.3 The admin can withdraw the allocated amounts

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The withdraw function allows the admin to withdraw any token and any amount of that token from the contract, this represents a significant issue, as it enables the admin to withdraw funds allocated to users, thereby disrupting the vesting functionality and preventing users from claiming their vested amounts.

Files Affected:

SHB.3.1: VestingModule.sol

```
function withdraw(IERC20Upgradeable tokenToWD, uint256 amount) public onlyRole(ADMIN_ROLE) returns (bool) {
    tokenToWD.safeTransfer(msg.sender, amount);
    return true;
}
```

Recommendation:

Consider adding a variable called totalAllocations which will contain the sum of allocations of all the users, and change the withdraw to the following implementation:

SHB.3.2: VestingModule.sol

```
function withdraw(IERC20Upgradeable tokenToWD, uint256 amount) public onlyRole(ADMIN_ROLE) returns (bool) {
    require(tokenToWD != token  totalAllocations + amount <= tokenToWD.balanceOf(address(this)),"Insuficient balance");
    tokenToWD.safeTransfer(msg.sender, amount);
    return true;
}
```

This way, all funds that are allocated to the users will be protected from being withdrawn by the admin.

Updates

The Kambria team resolved the issue by only allowing the owner to withdraw the funds that are not allocated to the vesting.

SHB.4 The initialize function can be front-run

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The contract initializes its state using an initialize function instead of a constructor to implement upgradability, leaving the initialization vulnerable to being front-run by an attacker.

Exploit Scenario:

The owner deploys the contract and performs the initialize function, then the attacker front-runs the transaction by paying a higher gas price and inputting malicious values into the contract.

Files Affected:

SHB.4.1: VestingModule.sol

```
function initialize(
    IERC20Upgradeable _token,
    uint256 _start,
    uint256 _duration
) public initializer {
    __Ownable_init();
    __UUPSUpgradeable_init();

    token = _token;
```

```
    start = _start;
    duration = _duration;
}
```

Recommendation:

Consider calling the initialize and the deployment of the contract in the same transaction, this can be done by using another contract, it can be either a proxy or a new contract, or consider adding access control to the function to prevent it from being called by an attacker.

Updates

The Kambria team resolved the issue by disabling the contract upgradaebility and moving the initialize logic to the constructor.

SHB.5 Owner can renounce ownership

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

Typically, the account that deploys the contract is also its owner. Consequently, the owner is able to engage in certain privileged activities in his own name. In smart contracts, the renounceOwnership function is used to renounce ownership, which means that if the contract's ownership has never been transferred, it will never have an Owner, rendering some owner-exclusive functionality unavailable.

Files Affected:

SHB.5.1: VestingModule.sol

```
contract VestingModule is Initializable, OwnableUpgradeable,
    ↪ UUPSUpgradeable, AccessControlUpgradeable{
```

Recommendation:

We recommend that you prevent the owner from calling renounceOwnership without first transferring ownership to a different address. Additionally, if you decide to use a multi-signature wallet, then the execution of the renounceOwnership will require for at least two or more users to be confirmed. Alternatively, you can disable Renounce Ownership functionality by overriding it.

Updates

The Kambria team resolved the issue by disabling the renounceOwnership function.

SHB.6 Unchecked arrays lengths

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The addAllocations function is used by the admin to allocate a specific amount to each one of the users, this function takes as arguments 2 different arrays, one for the addresses and the other one for the amounts allocated. However, this function does not check the lengths of the arguments to be the same, which can result in some cases a loss of elements from one of the arrays.

Files Affected:

SHB.6.1: VestingModule.sol

```
function addAllocations (
```

```
    address[] memory addresses,
    uint256[] memory allocations
) external onlyRole(ADMIN_ROLE) {
    for (uint256 i = 0; i < addresses.length; i++) {
        // Check if added allocation
        //uint256 totalAllocation_ = totalAllocation(addresses[i]);
        require(totalAllocation[addresses[i]] == 0, "VestingModule: There
           ↪  are a addressed has been added allocation already");
    }
     for (uint256 i = 0; i < addresses.length; i++) {
        totalAllocation[addresses[i]] = allocations[i];

    }

}
```

Recommendation:

It is recommended to verify the array arguments by making sure their lengths are equal.

Updates

The Kambria team resolved the issue by verifying the array arguments and making sure their lengths are equal.

SHB.7 Missing address and value verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The initialize function lacks a safety check in the address, the address-type argument _token should include a zero-address test, otherwise, the contract's functionality may become inaccessible. In addition to that, the function lacks a value safety check, the _start argument should be verified to be greater than the block.timestamp and the _duration argument should be verified to be different from zero.

Files Affected:

SHB.7.1: VestingModule.sol

```
function initialize(
    IERC20Upgradeable _token,
    uint256 _start,
    uint256 _duration
) public initializer {
    __Ownable_init();
    __UUPSUpgradeable_init();

    token = _token;
    start = _start;
    duration = _duration;
}
```

Recommendation:

We recommend that you verify the addresses and the values provided in the arguments. The issue can be addressed by utilizing require statements.

Updates

The Kambria team resolved the issue by verifying the addresses and the values provided in the arguments of the constructor.

SHB.8 Floating pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The contract makes use of the floating-point pragma 0.8.6. Contracts should be deployed using the same compiler version. Locking the pragma helps ensure that contracts will not unintentionally be deployed using another pragma, which in some cases may be an obsolete version, that may introduce issues to the contract system.

Files Affected:

SHB.8.1: VestingModule.sol

```
2102 pragma solidity ^0.8.6;
```

Recommendation:

Consider locking the pragma version. It is advised that floating pragma should not be used in production.

Updates

The Kambria team resolved the issue by locking the pragma version to 0.8.13.

4 Best Practices

BP.1 Remove duplicated checks

Description:

When calling the release or the releaseTo function, the block.timestamp is verified to be greater than the start timestamp, and the same check is duplicated inside the releasable function. It is recommended to remove the check from the release or the releaseTo functions.

Files Affected:

BP.1.1: VestingModule.sol

```
function release() external {
            uint256 releasable_ = releasable(msg.sender);

            require(releasable_ != 0, "VestingModule: Not eligible for
                ↪ release");
            require(
                block.timestamp > start,
                "VestingModule: The vesting has not started"
            );

            lastClaimedTimestamp[msg.sender] = block.timestamp;

            token.safeTransfer(msg.sender, releasable_);
    }
```

BP.1.2: VestingModule.sol

```
function releaseTo(address receiver) onlyRole(RELEASER_ROLE) external {
        uint256 releasable_ = releasable(receiver);

```

```
        require(releasable_ != 0, "VestingModule: Not eligible for release");
        require(
            block.timestamp > start,
            "VestingModule: The vesting has not started"
        );

        lastClaimedTimestamp[receiver] = block.timestamp;

        token.safeTransfer(receiver, releasable_);

    }
```

Status - Acknowledged

BP.2 Change the initialize function from public to external

Description:

Since the initialize function is only called from outside the contract, we recommend declaring it as external instead of public in order to optimize the gas.

Files Affected:

BP.2.1: VestingModule.sol

```
function initialize(
        IERC20Upgradeable _token,
        uint256 _start,
        uint256 _duration
    ) public initializer {
```

Status - Fixed

BP.3 Remove unnecessary loops

Description:

The addAllocations loops first on the addresses array to check if the address is already allocated, then loops again on the addresses array to map the addresses to their allocations. This action can be done in a single for loop, making the function more readable and optimized.

Files Affected:

BP.3.1: VestingModule.sol

```
function addAllocations (
    address[] memory addresses,
    uint256[] memory allocations
) external onlyRole(ADMIN_ROLE) {
    for (uint256 i = 0; i < addresses.length; i++) {
        // Check if added allocation
        //uint256 totalAllocation_ = totalAllocation(addresses[i]);
        require(totalAllocation[addresses[i]] == 0, "VestingModule: There
            ↪ are a addressed has been added allocation already");
    }
     for (uint256 i = 0; i < addresses.length; i++) {
        totalAllocation[addresses[i]] = allocations[i];

    }
```

Status - Fixed

5 Tests

Results:

→ Contract: VestingModule (18 passing)

✓ Should run faily the 'addAllocations' method

✓ Should return duration as 900

✓ Should return lastClaimedTimestamp as 0x00

✓ Should return owner address (41ms)

✓ Should return releasable as 0

✓ Should return start as 1676284624 (129ms)

✓ Should return token as 0xf8fd69502e81A545decf112c87704aD5283f5628

✓ Should return totalAllocation as 0

✓ Should return admin role as 0xa49807205ce4d355092ef5a8a18f56e8913cf4a201fbe287825b095693c21775

✓ Should return release role as 0x88f3509f0e42391f2d94ebfb2a37cbd0782b1b8f73715330017f4663290b8117

✓ Should return admin role as 0xa49807205ce4d355092ef5a8a18f56e8913cf4a201fbe287825b095693c21775

✓ Should return admin hasRole as true

✓ Should return true for the granted account as releaser (38ms)

✓ Should return false for the granted account as releaser (63ms)

✓ Should return false for the granted account as releaser (73ms)

✓ Should release sucessfully

✓ Should fail renounceOwnership

✓ Should transferOwnership to 0x2b8be9D1Bc04CE987E5A9eE55337fE50394f9D3B (108ms)

6 Conclusion

In this audit, we examined the design and implementation of Kambria's Vesting Module contract and discovered several issues of varying severity. Kambria team addressed all the issues raised in the initial report and implemented the necessary fixes.

However Shellboxes' auditors advised Kambria Team to maintain a high level of vigilance and participate in bounty programs in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
VestingModule.sol	2c4ff9707a6b6993b4220b9a430d9cdd

7.2 Re-Audit

Files	MD5 Hash
VestingModule.sol	3f9ecc52f158bf60db3a1f7383db4077

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

For a Contract Audit, contact us at contact@shellboxes.com