0x Guard

Smart contracts security assessment

Final report

Tariff: Standard

WEN

May 2023

0xguard.com

hello@0xguard.com

Contents

Introduction

The report has been prepared for **WEN**.

The WEN project is a vesting with an initial release amount of 10%.

The md5 sum of the files under investigation:

56af8d6f11f60fdb4a3906244c06f06b - WEN.sol

f18f5bc83ec88b6e8df7a41e152631d0 - WENAirdrop.sol

Report Update

The contracts code was updated according to this report.

The md5 sum of the updated files:

d4d29bf7f5b941c40c4ed944b0abb8d9 - WENAirdrop.sol

Name	WEN
Audit date	2023-05-11 - 2023-05-13
Language	Solidity
Platform	Ethereum

Contracts checked

Name	Address
WEN	0x830a8512db4F6fCA51968593E2667156C2c483A8
WENAirdrop	0x15c796d5575BFc8E8d7e285208c7ed4359507036

Procedure

We perform our audit according to the following procedure:

Automated analysis

- Scanning the project's smart contracts with several publicly available automated Solidity analysis tools
- Manual verification (reject or confirm) all the issues found by the tools

Manual audit

- Manually analyze smart contracts for security vulnerabilities
- Smart contracts' logic check

Known vulnerabilities checked

Title	Check result
Unencrypted Private Data On-Chain	passed
Code With No Effects	passed
Message call with hardcoded gas amount	passed
Typographical Error	passed
DoS With Block Gas Limit	passed
Presence of unused variables	passed
Incorrect Inheritance Order	passed
Requirement Violation	passed
Weak Sources of Randomness from Chain Attributes	passed
Shadowing State Variables	passed

Incorrect Constructor Name	passed
Block values as a proxy for time	passed
Authorization through tx.origin	passed
DoS with Failed Call	passed
Delegatecall to Untrusted Callee	passed
Use of Deprecated Solidity Functions	passed
Assert Violation	passed
State Variable Default Visibility	passed
Reentrancy	passed
Unprotected SELFDESTRUCT Instruction	passed
Unprotected Ether Withdrawal	passed
Unchecked Call Return Value	passed
Floating Pragma	passed
Outdated Compiler Version	passed
Integer Overflow and Underflow	passed
Function Default Visibility	passed

Classification of issue severity

High severity High severity issues can cause a significant or full loss of funds, change of contract ownership, major interference with contract logic. Such issues require immediate attention.

Medium severity Medium severity issues do not pose an immediate risk, but can be detrimental to the client's reputation if exploited. Medium severity issues may lead to a contract failure and can be fixed by modifying the contract state or redeployment. Such issues require attention.

Low severity Low severity issues do not cause significant destruction to the contract's functionality. Such issues are recommended to be taken into consideration.

Issues

High severity issues

No issues were found

Medium severity issues

1. Insufficient tokens for release (WENAirdrop)

Status: Fixed

The `release()` function performs transferring tokens to a user.

Because the variable `vestingSchedulesTotalAmount` is updated after the `require` check on L127, the contract needs more tokens (by the amount of vestedAmount) to run the function correctly.

You can also see this test case (`ISSUE1`) in the "Proof of concept" section.

Recommendation: Make sure the function works as intended.

Otherwise, consider changing the `require` statement on L127.

2. Vesting's initial amounts (WENAirdrop)

Status: Fixed

The `claim()` function releases and transfers 10% of the total user's tokens. Also, it creates vesting for the user (`msg.sender`). Such new vesting takes into account the initially released amount (_released=10%).

```
function claim(uint256 _amount, bytes32[] memory _proof) external nonReentrant {
    ...
    uint256 _released = _amount * 10/100;
```

```
        createVestingSchedule(msg.sender, _amount - _released, _released);
        IERC20(_token).transfer(msg.sender, _released);
        hasClaimed[msg.sender] = true;
    }

    function createVestingSchedule(address _beneficiary, uint256 _amount, uint256
_released) {
        ...
        vestingSchedules[_beneficiary] = VestingSchedule(
            true,
            block.timestamp,
            _amount,        // 90%
            _released
        );
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount + _amount;
    }
```

But at the same time total vesting amount is decreased for 10% (`_amount - _released`).

Example flow:

1. Alice expects to receive 200 tokens in total.

2. Alice calls the `claim()` function with `amount=200`:

-- Alice receives initial 20 tokens (10%).

-- Alice creates a new vesting with values `amountTotal=180, released=20`.

3. Alice waits until `duration` is expired and calls `release()`:

-- Now Alice can receive only 160 tokens (`amountTotal - released`).

Thus, after completing the vesting, Alice will only be able to receive 90% of `amountTotal`.

You can also see this test case (`ISSUE2`) in the "Proof of concept" section.

Also due to this issue 10% of vested token will be blocked in the contract.

Recommendation: We recommend creating a new vesting with `_amount` value without subtraction of `_released` value.

3. getWithdrawableAmount() underflow (WENAirdrop)

Status: Fixed

In the updated code the `getWithdrawableAmount()` function may revert with 'underflow' in amounts calculation. Thus it can block creating new vestings and executing the `withdraw()` function or revert the `getWithdrawableAmount()` function.

Such underflow happens due to incorrect updating of the `vestingSchedulesTotalAmount` state variable in the `createVestingSchedule()` function.

```
function createVestingSchedule(address _beneficiary, uint256 _amount, uint256 _released)
{
  ...
  vestingSchedules[_beneficiary] = VestingSchedule(
      true,
      block.timestamp,
      _amount,
      _released
  );
  vestingSchedulesTotalAmount = vestingSchedulesTotalAmount + _amount;
}
```

In the current code implementation 10% of `amountTotal` is transferred to the user, but not taken into account in the `released` variable (like in the `release()` function). So in this case `vestingSchedulesTotalAmount` will always be greater than the contract balance for the `release` amount.

Recommendation: We recommend subtracting the `_released` amount in the `createVestingSchedule()` function:

```
vestingSchedulesTotalAmount = vestingSchedulesTotalAmount + _amount - _released;
```

Low severity issues

1. Parameter validation (WENAirdrop)

Status: Fixed

Consider adding validation for the `duration_` parameter in the contract constructor.

2. Lack of require reason messages (WENAirdrop)

Status: Partially fixed

We recommend adding reason messages for the `require` statements on L61, L63, L124.

This will make debugging transactions easier.

Update: The reason message not specified in the `release()` function:

```
require(vestingSchedules[_address].initialized);
```

3. Documentation (WENAirdrop)

Status: Fixed

1. Consider adding a description for all constructor parameters.

2. We recommend fixing the description of the `claim()` function. Because it also allows claiming the initial amount.

Update: Point 2 has not been fixed.

4. Gas optimization (WENAirdrop)

Status: Fixed

1. The variable `_nft` is never used and can be removed.

   1. The variable `_duration` can be declared as '`immutable`'.

   2. The visibility of the `release()` function can be set to '`external`'.

3. The imported library `EnumerableSet` is never used and can be removed (L20, L28).

5. Lack of events (WENAirdrop)

Status: Fixed

We recommend adding events for the `setMerkleRoot()`, `setClaimState()`, `claim()`, `release()` functions to easily track changes off-chain.

Conclusion

WEN WEN, WENAirdrop contracts were audited. 3 medium, 5 low severity issues were found.
3 medium, 4 low severity issues have been fixed in the update.

We strongly recommend writing unit tests to have extensive coverage of the codebase minimize the possibility of bugs and ensure that everything works as expected.

The contract owner can disable the creation of new vestings at any time (by setting a new merkle-root or disabling the `_claimIsActive` value). Users interacting with the contract have to trust the owner.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without 0xGuard prior written consent.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts 0xGuard to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.
OxGuard retains exclusive publishing rights for the results of this audit on its website and social networks.

Proof of concept

For testing purposes we used a mock airdrop contract (without Merkle-proof functionality):

```
contract WENAirdropMock is Ownable, ReentrancyGuard {

    // using EnumerableSet for EnumerableSet.Bytes32Set;

    struct VestingSchedule {
        bool initialized;
        // start time of the vesting period
        uint256 start;
        // total amount of tokens to be released at the end of the vesting
        uint256 amountTotal;
        // amount of tokens released
        uint256 released;
    }

    // address of the ERC20 token
    address public immutable _token;
    // address of the MintPass
    address public immutable _nft;
    // duration of the vesting period in seconds
    uint256 public _duration;
    // merkle root for airdrop
    bytes32 public _merkleRoot;
    // mintpass status
    bool public _claimIsActive;
    // total amount vesting
    uint256 public vestingSchedulesTotalAmount;
    mapping(address => VestingSchedule) public vestingSchedules;
    mapping(address => bool) public hasClaimed;

    /**
     * @dev Creates a vesting contract.
     * @param token_ address of the ERC20 token contract
     */
    constructor(address token_, address nft_, uint256 duration_) {
        // Check that the token address is not 0x0.
        require(token_ != address(0x0));
        // Check that the token address is not 0x0.
        require(nft_ != address(0x0));
        // Set the token address.
        _token = token_;
        // Set the NFT address.
        _nft = nft_;
        // Set the duration.
        _duration = duration_;
    }

    function setMerkleRoot(bytes32 merkleRoot_) external onlyOwner {
        _merkleRoot = merkleRoot_;
    }

    function setClaimState(bool _isActive) external onlyOwner {
        _claimIsActive = _isActive;
    }

    /**
     * @notice Creates a new vesting schedule for a beneficiary.
     * @param _amount total amount of tokens to be released at the end of the vesting
     */
    function claim(uint256 _amount) external nonReentrant {
        // Airdrop must be active
        require(_claimIsActive, "Airdrop off");
        // Make sure the vesting schedule hasn't already been created
        require(!hasClaimed[msg.sender], "Already claimed");
        // Verify that the beneficiary is in the allowlist
        // bytes32 _leaf = keccak256(abi.encode(msg.sender, _amount));
        // require(MerkleProof.verify(_proof, _merkleRoot, _leaf), "Bad proof");
        // Release 10% of tokens now and create a vesting schedule for the rest
        uint256 _released = _amount * 10/100;
        createVestingSchedule(msg.sender, _amount - _released, _released);
        IERC20(_token).transfer(msg.sender, _released);
        hasClaimed[msg.sender] = true;
    }

    /**
     * @notice Creates a new vesting schedule for a beneficiary.
     * @param _beneficiary address of the beneficiary to whom vested tokens are
transferred
```

```
     * @param _amount total amount of tokens to be released at the end of the vesting
     * @param _released amount of tokens released
     */
    function createVestingSchedule(address _beneficiary, uint256 _amount, uint256
_released)
        internal
    {
        require(getWithdrawableAmount() >= _amount, "Insufficient tokens");
        require(_amount > 0, "Invalid amount");
        vestingSchedules[_beneficiary] = VestingSchedule(
            true,
            block.timestamp,
            _amount,        // 90
            _released       // 10
        );
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount + _amount;
    }

    /**
     * @notice Release vested amount of tokens.
     */
    function release() public nonReentrant {
        require(vestingSchedules[msg.sender].initialized);
        VestingSchedule storage vestingSchedule = vestingSchedules[msg.sender];
        uint256 vestedAmount = _computeReleasableAmount(vestingSchedule);
        require(getWithdrawableAmount() >= vestedAmount, "Insufficient tokens");
        vestingSchedule.released = vestingSchedule.released + vestedAmount;
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount - vestedAmount;
        IERC20(_token).transfer(msg.sender, vestedAmount);
    }

    /**
     * @dev Computes the releasable amount of tokens for a vesting schedule.
     * @return the amount of releasable tokens
     */
    function computeReleasableAmount(address _beneficiary)
        external
        view
        returns (uint256) {
        VestingSchedule storage vestingSchedule = vestingSchedules[_beneficiary];
        return _computeReleasableAmount(vestingSchedule);
    }

    function _computeReleasableAmount(VestingSchedule memory vestingSchedule)
        internal
        view
        returns (uint256) {
        // Retrieve the current time.
        uint256 currentTime = block.timestamp;
        if (currentTime >= vestingSchedule.start + _duration) {
            // If the current time is after the vesting period, all tokens are
releasable,
            // minus the amount already released.
            return vestingSchedule.amountTotal - vestingSchedule.released;
        } else {
            // Otherwise, some tokens are releasable.
            // Compute the number of seconds that have elapsed.
            uint256 vestedSeconds = currentTime - vestingSchedule.start;
            // Compute the proportion of time that has passed, in relation to the
vesting duration
            uint256 proportion = vestedSeconds * 1e18 / _duration;
            // Compute the amount of tokens that are vested.
            uint256 vestedAmount = (vestingSchedule.amountTotal * proportion) / 1e18;
            // Subtract the amount already released and return.
            if (vestedAmount < vestingSchedule.released) {
                return 0;
            }
            return vestedAmount - vestingSchedule.released;
        }
    }

    /**
     * @notice Withdraw the specified amount if possible.
     * @param amount the amount to withdraw
     */
    function withdraw(uint256 amount) external nonReentrant onlyOwner {
        require(getWithdrawableAmount() >= amount, "Insufficient funds");
        IERC20(_token).transfer(msg.sender, amount);
    }

    /**
     * @dev Returns the amount of tokens that can be withdrawn by the owner.
     * @return the amount of tokens
    */
    function getWithdrawableAmount() public view returns (uint256) {
        return IERC20(_token).balanceOf(address(this)) - vestingSchedulesTotalAmount;
    }
}
```

Test flow (check ISSUE1 and ISSUE2):

```
const { expect } = require("chai");
const {
  ethers: {
    getSigners,
    getContractFactory,
  },
} = require("hardhat");

describe("Vesting Amounts", function() {
  let owner;
  let alice;

  let wenContract;
  let wenAirdropContract;

  before(async() => {
    [owner, alice] = await getSigners();

    const wenFactory = await getContractFactory("WEN");
    wenContract = await wenFactory.deploy();
    await wenContract.deployed();

    const airdropFactory = await getContractFactory("WENAirdropMock");
    wenAirdropContract = await airdropFactory.deploy(wenContract.address,
wenContract.address, 0);
    await wenAirdropContract.connect(owner).setClaimState(true);
    await wenContract.connect(owner).transfer(wenAirdropContract.address, 200);
  });

  it("Should be 200 token on airdrop balance", async() => {
```

```
  expect(await wenContract.balanceOf(wenAirdropContract.address)).to.equal(200);
});

it("Should be 0 token on Alice balance", async() => {
  expect(await wenContract.balanceOf(alice.address)).to.equal(0);
});

describe("Alice claims", function () {
 before(async() => {
   await wenAirdropContract.connect(alice).claim(200);
 });
 it("Should increase Alice balance for 20 tokens (10% of 200)", async() => {
  expect(await wenContract.balanceOf(alice.address)).to.equal(20);
 });

 it("Should create new vesting with amountTotal=180, released=20", async() => {
  const newVesting = await wenAirdropContract.vestingSchedules(alice.address)
  // console.log("newVesting", newVesting)
  expect(newVesting.amountTotal).to.equal(180);
  expect(newVesting.released).to.equal(20);
 });

 describe(">>> ISSUE with release - Insufficient tokens", function () {
  it(">>> ISSUE1! It will revert due to Insufficient tokens", async() => {
   await expect(wenAirdropContract.connect(alice).release())
    .to.be.revertedWith("Insufficient tokens");
  });
 });

 describe("Alice releases all tokens after extra balance top-up", function () {
  before(async() => {
    // MOCK balance top-up
    await wenContract.connect(owner).transfer(wenAirdropContract.address, 160);

    // Release all tokens
    await wenAirdropContract.connect(alice).release();
  });

  it(">>> ISSUE2! Alice final balance only 20+160=180 tokens", async() => {
    expect(await wenContract.balanceOf(alice.address)).to.equal(180);
  });
```

```
      it("20 + 180 tokens in the airdrop contract", async() => {
          expect(await
wenContract.balanceOf(wenAirdropContract.address)).to.equal(20+160);
      });
    });
  });
});
```

Test output (check ISSUE1 and ISSUE2):

```
  Vesting Amounts
    ☒ Should be 200 token on airdrop balance
    ☒ Should be 0 token on Alice balance
    Alice claims
      ☒ Should increase Alice balance for 20 tokens (10% of 200)
      ☒ Should create new vesting with amountTotal=180, released=20
      >>> ISSUE with release - Insufficient tokens
        ☒ >>> ISSUE1! It will revert due to Insufficient tokens
      Alice releases all tokens after extra balance top-up
        ☒ >>> ISSUE2! Alice final balance only 20+160=180 tokens
        ☒ 20 + 180 tokens in the airdrop contract
```

Slither output

For the testing purposes we used a mock airdrop contract (without merleproof functionality):

```
contract WENAirdropMock is Ownable, ReentrancyGuard {

    // using EnumerableSet for EnumerableSet.Bytes32Set;

    struct VestingSchedule {
        bool initialized;
        // start time of the vesting period
        uint256 start;
        // total amount of tokens to be released at the end of the vesting
        uint256 amountTotal;
        // amount of tokens released
        uint256 released;
    }

    // address of the ERC20 token
    address public immutable _token;
    // address of the MintPass
    address public immutable _nft;
    // duration of the vesting period in seconds
    uint256 public _duration;
    // merkle root for airdrop
    bytes32 public _merkleRoot;
    // mintpass status
    bool public _claimIsActive;
    // total amount vesting
    uint256 public vestingSchedulesTotalAmount;
    mapping(address => VestingSchedule) public vestingSchedules;
    mapping(address => bool) public hasClaimed;

    /**
     * @dev Creates a vesting contract.
     * @param token_ address of the ERC20 token contract
     */
    constructor(address token_, address nft_, uint256 duration_) {
        // Check that the token address is not 0x0.
        require(token_ != address(0x0));
        // Check that the token address is not 0x0.
        require(nft_ != address(0x0));
        // Set the token address.
        _token = token_;
        // Set the NFT address.
        _nft = nft_;
        // Set the duration.
        _duration = duration_;
    }

    function setMerkleRoot(bytes32 merkleRoot_) external onlyOwner {
        _merkleRoot = merkleRoot_;
    }

    function setClaimState(bool _isActive) external onlyOwner {
        _claimIsActive = _isActive;
    }

    /**
     * @notice Creates a new vesting schedule for a beneficiary.
     * @param _amount total amount of tokens to be released at the end of the vesting
     */
    function claim(uint256 _amount) external nonReentrant {
        // Airdrop must be active
        require(_claimIsActive, "Airdrop off");
        // Make sure the vesting schedule hasn't already been created
        require(!hasClaimed[msg.sender], "Already claimed");
        // Verify that the beneficiary is in the allowlist
        // bytes32 _leaf = keccak256(abi.encode(msg.sender, _amount));
        // require(MerkleProof.verify(_proof, _merkleRoot, _leaf), "Bad proof");
        // Release 10% of tokens now and create a vesting schedule for the rest
        uint256 _released = _amount * 10/100;
        createVestingSchedule(msg.sender, _amount - _released, _released);
        IERC20(_token).transfer(msg.sender, _released);
        hasClaimed[msg.sender] = true;
    }

    /**
     * @notice Creates a new vesting schedule for a beneficiary.
     * @param _beneficiary address of the beneficiary to whom vested tokens are
 transferred
```

```
     * @param _amount total amount of tokens to be released at the end of the vesting
     * @param _released amount of tokens released
     */
    function createVestingSchedule(address _beneficiary, uint256 _amount, uint256 
_released)
        internal
    {
        require(getWithdrawableAmount() >= _amount, "Insufficient tokens");
        require(_amount > 0, "Invalid amount");
        vestingSchedules[_beneficiary] = VestingSchedule(
            true,
            block.timestamp,
            _amount,        // 90
            _released       // 10
        );
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount + _amount;
    }

    /**
     * @notice Release vested amount of tokens.
     */
    function release() public nonReentrant {
        require(vestingSchedules[msg.sender].initialized);
        VestingSchedule storage vestingSchedule = vestingSchedules[msg.sender];
        uint256 vestedAmount = _computeReleasableAmount(vestingSchedule);
        require(getWithdrawableAmount() >= vestedAmount, "Insufficient tokens");
        vestingSchedule.released = vestingSchedule.released + vestedAmount;
        vestingSchedulesTotalAmount = vestingSchedulesTotalAmount - vestedAmount;
        IERC20(_token).transfer(msg.sender, vestedAmount);
    }

    /**
     * @dev Computes the releasable amount of tokens for a vesting schedule.
     * @return the amount of releasable tokens
     */
    function computeReleasableAmount(address _beneficiary)
        external
        view
        returns (uint256) {
        VestingSchedule storage vestingSchedule = vestingSchedules[_beneficiary];
        return _computeReleasableAmount(vestingSchedule);
    }

    function _computeReleasableAmount(VestingSchedule memory vestingSchedule)
        internal
        view
        returns (uint256) {
        // Retrieve the current time.
        uint256 currentTime = block.timestamp;
        if (currentTime >= vestingSchedule.start + _duration) {
            // If the current time is after the vesting period, all tokens are
releasable,
            // minus the amount already released.
            return vestingSchedule.amountTotal - vestingSchedule.released;
        } else {
            // Otherwise, some tokens are releasable.
            // Compute the number of seconds that have elapsed.
            uint256 vestedSeconds = currentTime - vestingSchedule.start;
            // Compute the proportion of time that has passed, in relation to the
vesting duration
            uint256 proportion = vestedSeconds * 1e18 / _duration;
            // Compute the amount of tokens that are vested.
            uint256 vestedAmount = (vestingSchedule.amountTotal * proportion) / 1e18;
            // Subtract the amount already released and return.
            if (vestedAmount < vestingSchedule.released) {
                return 0;
            }
            return vestedAmount - vestingSchedule.released;
        }
    }

    /**
     * @notice Withdraw the specified amount if possible.
     * @param amount the amount to withdraw
     */
    function withdraw(uint256 amount) external nonReentrant onlyOwner {
        require(getWithdrawableAmount() >= amount, "Insufficient funds");
        IERC20(_token).transfer(msg.sender, amount);
    }

    /**
     * @dev Returns the amount of tokens that can be withdrawn by the owner.
     * @return the amount of tokens
    */
    function getWithdrawableAmount() public view returns (uint256) {
        return IERC20(_token).balanceOf(address(this)) - vestingSchedulesTotalAmount;
    }
}
```

Test flow (check ISSUE1 and ISSUE2):

```
const { expect } = require("chai");
const {
  ethers: {
    getSigners,
    getContractFactory,
  },
} = require("hardhat");

describe("Vesting Amounts", function() {
  let owner;
  let alice;

  let wenContract;
  let wenAirdropContract;

  before(async() => {
    [owner, alice] = await getSigners();

    const wenFactory = await getContractFactory("WEN");
    wenContract = await wenFactory.deploy();
    await wenContract.deployed();

    const airdropFactory = await getContractFactory("WENAirdropMock");
    wenAirdropContract = await airdropFactory.deploy(wenContract.address,
wenContract.address, 0);
    await wenAirdropContract.connect(owner).setClaimState(true);
    await wenContract.connect(owner).transfer(wenAirdropContract.address, 200);
  });

  it("Should be 200 token on airdrop balance", async() => {
      expect(await wenContract.balanceOf(wenAirdropContract.address)).to.equal(200);
  });

  it("Should be 0 token on Alice balance", async() => {
      expect(await wenContract.balanceOf(alice.address)).to.equal(0);
  });

  describe("Alice claims", function () {
    before(async() => {
        await wenAirdropContract.connect(alice).claim(200);
    });
    it("Should increase Alice balance for 20 tokens (10% of 200)", async() => {
      expect(await wenContract.balanceOf(alice.address)).to.equal(20);
    });

    it("Should create new vesting with amountTotal=180, released=20", async() => {
      const newVesting = await wenAirdropContract.vestingSchedules(alice.address)
      // console.log("newVesting", newVesting)
      expect(newVesting.amountTotal).to.equal(180);
      expect(newVesting.released).to.equal(20);
    });

    describe(">>> ISSUE with release - Insufficient tokens", function () {
      it(">>> ISSUE1! It will revert due to Insufficient tokens", async() => {
        await expect(wenAirdropContract.connect(alice).release())
          .to.be.revertedWith("Insufficient tokens");
      });
    });

    describe("Alice releases all tokens after extra balance top-up", function () {
      before(async() => {
          // MOCK balance top-up
          await wenContract.connect(owner).transfer(wenAirdropContract.address, 160);

          // Release all tokens
          await wenAirdropContract.connect(alice).release();
      });

      it(">>> ISSUE2! Alice final balance only 20+160=180 tokens", async() => {
          expect(await wenContract.balanceOf(alice.address)).to.equal(180);
      });
```

```
      it("20 + 180 tokens in the airdrop contract", async() => {
          expect(await
wenContract.balanceOf(wenAirdropContract.address)).to.equal(20+160);
      });
    });
  });
});
```

Test output (check ISSUE1 and ISSUE2):

```
  Vesting Amounts
    ⊠ Should be 200 token on airdrop balance
    ⊠ Should be 0 token on Alice balance
    Alice claims
      ⊠ Should increase Alice balance for 20 tokens (10% of 200)
      ⊠ Should create new vesting with amountTotal=180, released=20
      >>> ISSUE with release - Insufficient tokens
        ⊠ >>> ISSUE1! It will revert due to Insufficient tokens
      Alice releases all tokens after extra balance top-up
        ⊠ >>> ISSUE2! Alice final balance only 20+160=180 tokens
        ⊠ 20 + 180 tokens in the airdrop contract
```

0x Guard