



coinspect
You build, we defend.



Smart Contract Audit
Green Minting Token
August, 2025



Green Minting Token Smart Contract Audit

Version: v250813

Prepared for: Green Mint

August 2025

Security Assessment

1. Executive Summary
2. Summary of Findings
 - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
 - 4.1 Security assumptions
 - 4.2 Decentralization
 - 4.3 Testing
 - 4.4 Code quality
5. Detailed Findings

GRM-01 - Funds are permanently locked after vesting schedule completes

GRM-02 - Late token supply skews the vesting schedule

GRM-03 - Off-by-one error in stage calculation causes premature unlocking

GRM-04 - Missing constructor validation can lead to funds lock and vesting errors




GRM-05 - Missing zero address checks in constructor can break contract functionality

GRM-06 - Unchecked token transfer can lead to silent failures and accounting inconsistencies

6. Disclaimer

1. Executive Summary

In **August, 2025**, **Green Mint** engaged **Coinspect** to perform a Smart Contract Audit of Green Mint's Green Minting Token. The objective of the project was to evaluate the security of the smart contracts.

 Solved	 Caution Advised	 Resolution Pending
High 2	High 0	High 0
Medium 2	Medium 0	Medium 0
Low 0	Low 0	Low 0
No Risk 2	No Risk 0	No Risk 0
Total 6	Total 0	Total 0

Coinspect identified the following issues in the Green Minting Token's logic. High severity findings include permanent fund lock-up after vesting completion (GRM-01) and distorted vesting slope due to delayed token supply (GRM-02). Medium severity issues involve premature unlocking from an off-by-one stage miscalculation (GRM-03) and missing constructor checks that risk incorrect vesting behavior (GRM-04).

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
GRM-01	Funds are permanently locked after vesting schedule completes	High
GRM-02	Late token supply skews the vesting schedule	High
GRM-03	Off-by-one error in stage calculation causes premature unlocking	Medium
GRM-04	Missing constructor validation can lead to funds lock and vesting errors	Medium
GRM-05	Missing zero address checks in constructor can break contract functionality	None
GRM-06	Unchecked token transfer can lead to silent failures and accounting inconsistencies	None

3. Scope

The scope was set to be the repository:

- <https://github.com/green-minting/mint-token> at commit `a7180aa0e94dc563d0c7967d10cc563eb9120dee`.

4. Assessment

On this engagement, Coinspect reviewed the project composed of two Solidity smart contracts: `VestedLock` and `GreenMintingToken`.

The `GreenMintingToken` contract is an `ERC20` token implementing the `EIP-3009` standard, which facilitates gas-less transactions through signed off-chain messages. It also inherits burnable functionalities from `OpenZeppelin`.

The `VestedLock` contract is designed to manage a time-based vesting schedule for a single designated beneficiary account (`vestingAccount`). It releases portions of the `GreenMintingToken` according to a predefined schedule of percentages and time intervals.

EIP3009

This abstract contract provides a standard implementation for `EIP-3009`. It uses `EIP-712` for typed data hashing to protect against replay attacks across different chains or contracts. It enables three main actions via off-chain signatures:

- `_transferWithAuthorization`: Allows a third party to submit a transfer transaction on behalf of a token holder.
- `_receiveWithAuthorization`: Allows a recipient to submit a transaction to receive tokens, authorized by the sender's signature. This includes a protection where `msg.sender` must be the recipient (`to`) to prevent front-running.
- `_cancelAuthorization`: Allows a user to cancel a signed authorization nonce before it is used. It correctly manages authorization states to prevent nonce reuse and includes checks for the validity period of signatures (`validAfter` and `validBefore`).

GreenMintingToken

Implements an `ERC20` token named "Green Minting Token" (`MINT`). It inherits from `OpenZeppelin's ERC20Burnable` and the provided `EIP3009` contract. During construction, it mints an initial supply to a list of pre-funded accounts and allocates a specified `vestedAmount` to the contract deployer. It exposes the functions for the `EIP-3009` implementation. No further minting capabilities are available post-deployment.

VestedLock

Locks tokens and releases them over time to a single `vestingAccount`. The vesting schedule is defined by three immutable parameters:

- `unvestingStartTimestamp`: The timestamp when the vesting period begins.
- `secPerStage`: The duration of each vesting stage.
- `claimingPercentsSchedule`: An array defining the percentage of the total vested tokens that becomes claimable at each stage.

The `availableVestedTokens()` function calculates the amount of tokens currently claimable by determining the current stage and summing the corresponding percentages from the schedule.

4.1 Security assumptions

For this security assessment, Coinspect made the following assumptions:

1. The deployer of the contracts is trusted to provide correct and non-malicious constructor parameters (e.g., vesting schedule, initial token allocations).
2. Users (token holders) are responsible for securely managing the private keys used to sign [EIP-3009](#) messages. Compromise of a user's key will allow an attacker to authorize transactions on their behalf.

4.2 Decentralization

The protocol incorporates elements of both centralization and decentralization.

Roles and Privileges:

- **Deployer**: The account that deploys the `GreenMintingToken` contract has significant initial power. It determines all pre-funded accounts and their amounts, and receives the entire `vestedAmount`. The deployer of the `VestedLock` contract sets all vesting parameters, including the beneficiary `vestingAccount`, which are immutable.

- **Vesting Account:** The `vestingAccount` in the `VestedLock` contract is the only address permitted to call the `claimVestedTokens()` function. This is a privileged role, which is limited to withdrawing its own vested funds.

Ownership

Neither contract implements a standard ownership pattern (e.g., `Ownable`). After deployment, there is no single privileged account that can alter the contracts' logic, pause functionality, or upgrade them. All parameters in `VestedLock` are immutable, enhancing trust and predictability.

The `vestingAccount` could be set to a multisig wallet address, which would decentralize control over the claiming of vested tokens. Users should be aware that if the `vestingAccount` is an Externally Owned Account (EOA), its control rests with a single private key.

4.3 Testing

The tests suite provides coverage of the primary functionalities of the `VestedLock` and `GreenMintingToken` contracts. A deployment script was also provided, which confirms the intended production configuration aligns with a full 100% vesting schedule.

The test suite is structured and covers the following areas:

- **Initial State:** Tests verify that the `GreenMintingToken` is minted to pre-funded accounts as expected and that the `VestedLock` contract is properly funded with the total amount of tokens designated for vesting.
- **Access Control:** The suite confirms that only the designated `vestingAccount` can successfully call the `claimVestedTokens` function, while attempts from other accounts are correctly reverted.
- **Time-based Logic:** The tests use Hardhat's `evm_mine` functionality to manipulate block timestamps, confirming that vesting claims are prohibited before the `unvestingStartTimestamp` and are permitted afterward.
- **EIP-3009 Functionality:** The test cases for the EIP-3009 implementation are comprehensive and validate:
 1. The standard `transferWithAuthorization` flow, where a third party can submit a transaction on behalf of a signer.
 2. The `cancelAuthorization` flow, ensuring a signed nonce can be invalidated before use.

3. An adversarial scenario for `receiveWithAuthorization`, confirming that only the designated recipient (`to`) can execute the transaction, which mitigates front-running attacks.

While the test suite is robust, Coinspect has identified an area for potential improvement to further increase confidence in the contract's behavior under all conditions:

- Cumulative Vesting Claims: the test named "Unvest tokens in schedule" validates the claiming process on a stage-by-stage basis. However, it does not cover the scenario where a user forgoes claiming for several stages and then attempts to claim the total accumulated amount from all missed stages at once. Coinspect recommends adding a test case that advances time across multiple vesting stages before the first claim is made. This would explicitly verify that the cumulative logic within the `availableVestedTokens` function is calculated correctly in such a scenario.

4.4 Code quality

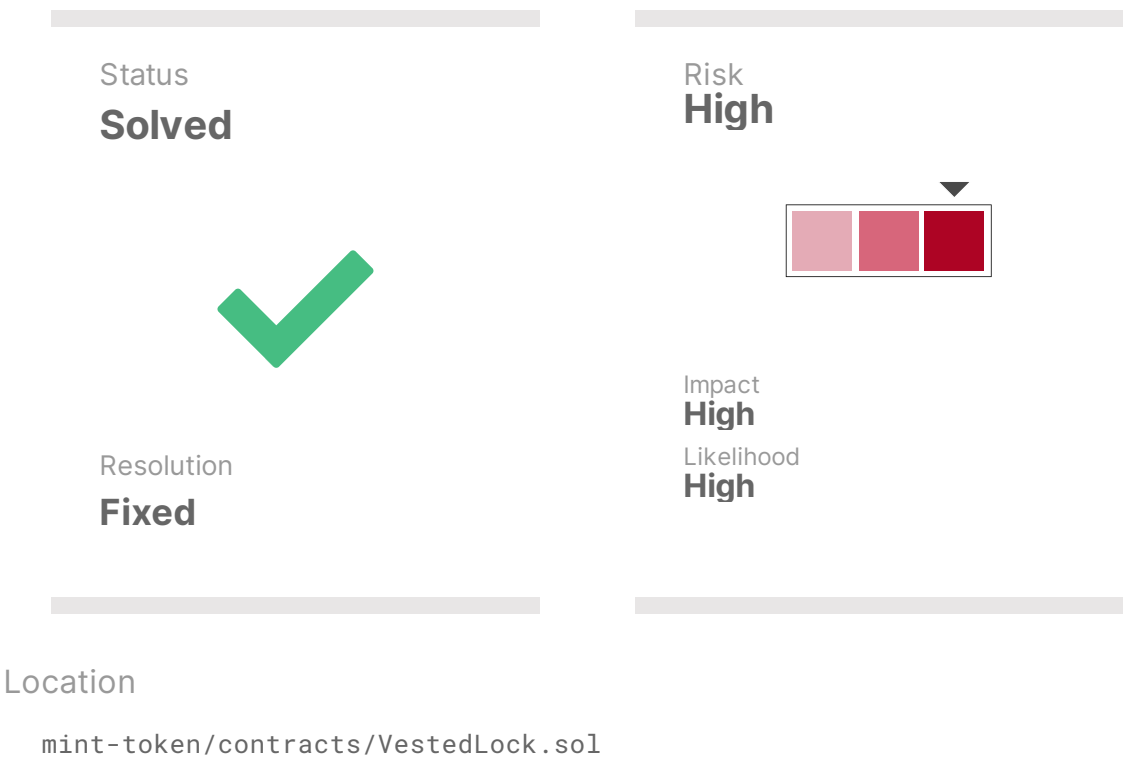
The overall code quality is good, adhering to common Solidity practices.

- Documentation and Comments: The `EIP3009` contract contains good Natspec documentation, explaining the purpose of functions and parameters. The `VestedLock` and `GreenMintingToken` contracts, however, lack Natspec comments. Inline comments are used sparingly. Coinspect considers that adding comprehensive Natspec documentation to all public and external functions would significantly improve the clarity and maintainability of the codebase.
- Clarity and Structure: The code is well-organized, and variable names are descriptive. The use of custom errors (e.g., `EIP3009_AuthorizationExpired`) instead of require strings is a modern and gas-efficient practice.
- Best Practices: The contracts extend libraries from OpenZeppelin, a widely audited and secure source. The checks-effects-interactions pattern is correctly implemented in `VestedLock.claimVestedTokens`, where the state (`claimedVestedAmount`) is updated before the external call (`token.transfer`), mitigating reentrancy risks. The use of an immutable `DENOMINATOR` for percentage calculations is a safe and standard approach for handling fractional arithmetic.

5. Detailed Findings

GRM-01

Funds are permanently locked after vesting schedule completes



Description

The contract becomes completely unusable and all remaining funds are permanently locked due to an array out-of-bounds vulnerability in the `availableVestedTokens()` function.

The contract implements a time-based vesting mechanism where the `claimingPercentsSchedule` array represents different time periods during which

specific percentages of tokens become available for claiming. Each element in the array corresponds to a vesting stage and the `secPerStage` parameter defines the duration of each stage. The contract calculates the current stage based on the elapsed time since `unvestingStartTimestamp` to determine how much of the vesting schedule has unlocked.

When `block.timestamp` reaches or exceeds the expected end of the vesting schedule, the calculated `stage` value will exceed the length of the `claimingPercentsSchedule` array.

When the loop attempts to access `claimingPercentsSchedule[i]` where `i >= claimingPercentsSchedule.length`, it triggers an array out-of-bounds error, causing the transaction to revert.

```
function availableVestedTokens() public view returns (uint256) {
    //...
    uint256 stage = ((block.timestamp - unvestingStartTimestamp) /
        secPerStage) + 1;
    uint256 availablePercents = 0;

    for (uint i = 0; i < stage; i++) {
        availablePercents += claimingPercentsSchedule[i];
    }
    //...
}
```

Once the vesting period is complete and `stage` exceeds the array length, any call to `availableVestedTokens()` will revert. Since `claimVestedTokens()` depends on `availableVestedTokens()`, it also becomes unusable. This creates a permanent denial of service where remaining tokens become locked in the contract and there is no recovery mechanism in place.

Coinspect considers this issue to have a high likelihood as it will inevitably occur for any contract that remains active beyond its intended vesting schedule. The impact is high as it results in permanent fund loss.

Recommendation

Ensure that the `stage` value used in the loop is the minimum between the calculated `stage` value and `claimingPercentsSchedule.length` to prevent array out-of-bounds access.

Status

Fixed on commit `7db8644fb463abd435b78645ee163c3ce2f9a626`.

A boundary check was added to prevent `stage` from exceeding `claimingPercentsSchedule.length`. When the vesting schedule is complete, the function now returns `leftVestedAmount` instead.

Proof of concept

The following test demonstrates the vulnerability using a 2-stage vesting schedule (20%, 80%):

```
import { expect } from "chai";
import hre, { ethers } from "hardhat";

describe("VestedLock Array Out-of-Bounds PoC", function () {
  it("Vesting reverts when accessing beyond vesting schedule", async () => {
    const [deployer, vestingAccount] = await hre.ethers.getSigners();

    // Deploy contracts
    const GreenMintingToken = await
hre.ethers.getContractFactory("GreenMintingToken");
    const VestedLock = await
hre.ethers.getContractFactory("VestedLock");

    const vestedAmount = BigInt(100000);
    const greenMintingToken = await GreenMintingToken.deploy([], [],
vestedAmount);

    // Create SHORT vesting schedule: only 2 stages (20%, 80%)
    const claimingPercentsSchedule = [2000, 8000];
    const secPerStage = 10;

    const vestedLock = await VestedLock.deploy(
      vestingAccount,
      secPerStage,
      claimingPercentsSchedule,
      (await ethers.provider.getBlock("latest")).!.timestamp,
      greenMintingToken
    );

    await greenMintingToken.transfer(vestedLock, vestedAmount);

    console.log("Schedule length:", claimingPercentsSchedule.length,
"stages");
    console.log("Schedule:", claimingPercentsSchedule);

    // Stage 1 & 2: Normal operation ✓
    await hre.network.provider.send("evm_increaseTime", [1]);
    await hre.network.provider.send("evm_mine");
    let available = await vestedLock.availableVestedTokens();
    console.log("Stage 1 available:", available.toString());

    await hre.network.provider.send("evm_increaseTime", [secPerStage]);
    await hre.network.provider.send("evm_mine");
    available = await vestedLock.availableVestedTokens();
    console.log("Stage 2 available:", available.toString());
```

```

// Stage 3: VULNERABILITY TRIGGERED ✕
console.log("\nTrigger vulnerability: Advancing to stage 3...");
await hre.network.provider.send("evm_increaseTime", [secPerStage]);
await hre.network.provider.send("evm_mine");

// Test that availableVestedTokens() reverts
let availableTokensReverted = false;
try {
  await vestedLock.availableVestedTokens();
} catch (error: any) {
  console.log("✓ availableVestedTokens() reverted:",
error.message.includes("0x32"));
  availableTokensReverted = true;
}

// Test that claimVestedTokens() also reverts
let claimReverted = false;
try {
  await vestedLock.connect(vestingAccount).claimVestedTokens();
} catch (error: any) {
  console.log("✓ claimVestedTokens() reverted:",
error.message.includes("0x32"));
  claimReverted = true;
}

// Verify both functions are broken
expect(availableTokensReverted).to.be.true;
expect(claimReverted).to.be.true;

  console.log("  Contract unusable after vesting schedule
  completes");
});
});

```

Test Output:

```

VestedLock Array Out-of-Bounds PoC
Schedule length: 2 stages
Schedule: [ 2000, 8000 ]
Stage 1 available: 20000
Stage 2 available: 100000

Trigger vulnerability: Advancing to stage 3...
✓ availableVestedTokens() reverted: true
✓ claimVestedTokens() reverted: true
  Contract unusable after vesting schedule completes
    ✓ Vesting reverts when accessing beyond vesting schedule

```

The test confirms that once the vesting schedule completes and time advances beyond the final stage, both `availableVestedTokens()` and `claimVestedTokens()` functions become permanently unusable due to array out-of-bounds access, leaving all remaining funds permanently locked.

GRM-02

Late token supply skews the vesting schedule

Status

Solved



Resolution

Fixed

Risk

High



Impact

High

Likelihood

High

Location

`mint-token/contracts/VestedLock.sol`

Description

The vesting logic in `VestedLock` incorrectly ties the vesting slope to the contract's current token balance, allowing the beneficiary to claim a disproportionate share of tokens when funding occurs late. This behavior enables early full claims despite only partial vesting time elapsing, undermining the intended time-based release schedule. The root cause is the use of a dynamically computed vested amount, which includes tokens supplied at any time—even after several vesting stages have passed.

In the `availableVestedTokens()` function:

```
uint256 fullVestedAmount = token.balanceOf(address(this)) +  
claimedVestedAmount;
```

```
uint256 availableToClaim = ((fullVestedAmount * availablePercents) /
DENOMINATOR) - claimedVestedAmount;
```

The value of `fullVestedAmount` reflects the current token balance and does not distinguish when tokens were supplied. If tokens are deposited after some vesting stages have elapsed, the `availablePercents` will include past stages, and the newly supplied tokens will be considered fully vestable as if they had been present since the beginning.

For example, with a 3-stage schedule of `[3000, 2000, 5000]` and `secPerStage = 10`, if the contract is funded after 20 seconds (two full stages elapsed) with 100,000 tokens, the function will compute:

```
availablePercents = 3000 + 2000 + 5000 = 10000; // due to +1 in stage
calculation
fullVestedAmount = 100000;
availableToClaim = 100000 * 10000 / 10000 - 0 = 100000;
```

The beneficiary will be able to claim 100% of the vested tokens even though only 20 seconds have passed.

This violates the assumption of a time-locked release and creates an incentive to delay token funding in order to accelerate access to the full vested amount.

Coinspect considers the likelihood and impact of this issue to be high, as the vesting logic can be manipulated (intentionally or inadvertently) by deferring token funding, resulting in premature access to the full allocation and a fundamental breach of time-based vesting guarantees.

Recommendation

The total vested amount must be fixed at deployment or first funding and stored immutably. The `availableVestedTokens()` function should compute against this static reference, rather than using the live balance.

Status

Fixed on commit `7db8644fb463abd435b78645ee163c3ce2f9a626`.

Added a `lockFunds()` function that allows the owner to fund the contract only once, storing the locked amount in `leftVestedAmount`. The vesting calculation now uses this fixed amount `leftVestedAmount` instead of the current token balance.

The `lockFunds` function should be called before `unvestingStartTimestamp` to ensure the vesting schedule works as expected.

On commit `0db99a2139e1e2a9d4b43ee9d66c7bc8501046fb`, the Team modified the constructor to enforce setting the `fullVestedAmount` when deploying the smart contract.

Proof of Concept

The following test shows how users are allowed to prematurely claim their tokens if they are supplied after the vesting period starts.

```
it("Vesting slope is altered by late token funding", async () => {
  const [
    deployer,
    -,
    --,
    vestingAccount,
    ...otherAccounts
  ] = await hre.ethers.getSigners();

  const GreenMintingToken = await
hre.ethers.getContractFactory("GreenMintingToken");
  const VestedLock = await
hre.ethers.getContractFactory("VestedLock");

  const vestedAmount = BigInt(100_000);
  const claimingPercentsSchedule = [3000, 2000, 5000]; // 100%
  const secPerStage = 10;

  const tokenHolderA = otherAccounts[0];
  const tokenHolderB = otherAccounts[1];

  const greenMintingToken = await GreenMintingToken.deploy(
    [tokenHolderA.address, tokenHolderB.address],
    [BigInt(0), BigInt(0)],
    vestedAmount
  );

  const currentBlock = await ethers.provider.getBlock("latest");
  const unvestStartTimestamp = currentBlock!.timestamp;

  const vestedLock = await VestedLock.deploy(
    vestingAccount.address,
    secPerStage,
    claimingPercentsSchedule,
    unvestStartTimestamp,
    greenMintingToken
  );

  console.log(`\n Vesting Contract Deployed`);
  console.log(`    Unvesting starts at:
${unvestStartTimestamp}`);
  console.log(`    ⚡ Stage duration:           ${secPerStage}
```

```

seconds`);
    console.log(`    Tokens vested (not yet transferred):
    ${vestedAmount}`);

    // Fast forward past 2 full stages
    const targetTimestamp = unvestStartTimestamp + 2 * secPerStage;
    await mineBlocks(targetTimestamp);

    const afterMine = await ethers.provider.getBlock("latest");
    console.log(`\n    Time Travel`);
    console.log(`    Current block timestamp:
    ${afterMine.timestamp}`);
    console.log(`    Time since start:                ${afterMine.timestamp
    - unvestStartTimestamp} seconds`);

    // Before funding
    const balanceBefore = await greenMintingToken.balanceOf(await
vestedLock.getAddress());
    console.log(`\n    Funding Check`);
    console.log(`    Token balance before funding: ${balanceBefore}`);

    // ✓ Fund the contract *after* two vesting stages have passed
    await greenMintingToken.transfer(await vestedLock.getAddress(),
vestedAmount);

    const balanceAfter = await greenMintingToken.balanceOf(await
vestedLock.getAddress());
    console.log(`    ✓ Token balance after funding: ${balanceAfter}`);

    // Check available tokens now
    const availableNow = await vestedLock.availableVestedTokens();

    const unlockedStages = 3; // This is what the contract thinks due
to stage = ... + 1
    const expectedUnlockedStages = 2;
    const unlockedPercent = claimingPercentsSchedule
        .slice(0, expectedUnlockedStages)
        .reduce((a, b) => a + b, 0);
    const expectedClaim = (vestedAmount * BigInt(unlockedPercent)) /
    BigInt(10_000);

    console.log(`\n    Slope Debug`);
    console.log(`    Stages expected unlocked:
    ${expectedUnlockedStages}`);
    console.log(`    Claimed percent (expected):    ${unlockedPercent
    / 100}%`);
    console.log(`    Expected unlocked tokens:
    ${expectedClaim}`);
    console.log(`    ! Actual available tokens:
    ${availableNow}`);
    console.log(`    Mismatch is EXPECTED in this PoC: shows late
    funding breaks slope.`);

    // Failing on purpose to prove the bug
    expect(availableNow).to.equal(expectedClaim); // <- this should
    FAIL

    // Unreachable
    // Try to claim
    await



```

```
expect(vestedLock.connect(vestingAccount).claimVestedTokens()).to.not.reverted;

    const received = await
greenMintingToken.balanceOf(vestingAccount.address);
    console.log(`\n Claim Result`);
    console.log(`      Tokens claimed:                ${received}`);
    console.log(`      Should have been:
${expectedClaim}`);
});
```

GRM-03

Off-by-one error in stage calculation causes premature unlocking

Status Solved	Risk Medium
	
Resolution Fixed	Impact Medium Likelihood High
Location <code>mint-token/contracts/VestedLock.sol</code>	

Description

The vesting stage computation introduces an off-by-one error by incrementing the stage index before any time has elapsed, resulting in premature unlocking of tokens at the very start of the vesting schedule. This leads to a misalignment between the intended lock period and the actual release behavior, where the first tranche becomes available immediately at `unvestingStartTimestamp`.

In `availableVestedTokens()`, the stage is computed as:

```
uint256 stage = ((block.timestamp - unvestingStartTimestamp) /  
secPerStage) + 1;
```

This formula adds 1 to the result of the division, causing the first vesting stage to be considered completed at the moment vesting begins. For example, if `block.timestamp == unvestingStartTimeStamp`, the computed stage is 1, which incorrectly unlocks the first scheduled percentage:

```
// At timestamp == unvestingStartTimeStamp
stage = ((0) / secPerStage) + 1 = 1
// → availablePercents includes claimingPercentsSchedule[0]
```

As a result, tokens are partially unlocked even though no time has passed, which violates typical vesting semantics where the first release is expected after the first full stage duration.

Coinspect considers the likelihood of this issue to be high and the impact to be medium, especially in systems that enforce strict vesting timelines. While the behavior may be unintentionally missed during testing, it will reliably result in premature token release under production conditions.

Recommendation

Tie the result to the length of the `claimingPercentsSchedule` array to avoid out-of-bounds access. If a grace period is desired before the first unlock, it should be enforced explicitly rather than introduced as a side effect of the formula.

Status

Fixed.

The Team responded that by design, the first stage of vesting is intended to unlock immediately when `block.timestamp >= unvestingStartTimeStamp` — that is, at the beginning of the first interval.

However Coinspect would like to point out that this behavior only works if the `unvestingStartTimeStamp` is set at least at `now + stage duration`. Otherwise users would be able to claim all the tokens at $(N^{\circ}\text{stages} - 1) * \text{stage duration}$. This has significant impact when the stage duration is long enough, e.g. a year.

Coinspect made a proof of concept that claims tokens on each stage showing that the `vestingAccount` bypasses the need to wait for all periods to pass.

```
Off-by-One Vesting Stage Walkthrough (Logging Only)
Unvesting Start: 1754485618
Total amount: 90000
Stage duration: 10 seconds
Vesting schedule: [3000, 2000, 5000]

Stage 0 Check
Time elapsed since vesting start: 2s
Expected unlocked percent: 0%
Expected unlocked amount: 0
Actual available amount: 27000

Stage 1 Check
Time elapsed since vesting start: 10s
Expected unlocked percent: 30%
Expected unlocked amount: 27000
Actual available amount: 45000

Stage 2 Check
Time elapsed since vesting start: 20s
Expected unlocked percent: 50%
Expected unlocked amount: 45000
Actual available amount: 90000
All tokens are expected only after 30s
But full unlock occurs after only 20s if bug is present
```

On commit `b71e7aef6b8705b23b42e42d5e5f8d06393a3fb7`, the Team removed the off-by-one stage calculation and replaced it with a functionally equivalent implementation. The Team clarified that this behavior is intentional, as the vesting schedule is designed to release an initial portion of tokens immediately when the schedule starts. The intended release plan was provided as follows:

```
Available at ICO: 30%
1 year after ICO: 20%
2 years after ICO: 10%
3 years after ICO: 5%
4 years after ICO: 5%
5 years after ICO: 5%
6 years after ICO: 5%
7 years after ICO: 5%
8 years after ICO: 5%
9 years after ICO: 5%
10 years after ICO: 2.5%
11 years after ICO: 2.5%
```

Proof of Concept

The following test shows how a user is able to claim the portion of the first stage immediately after the vesting period starts.

```

    it("Immediately unlocks first vesting stage due to off-by-one
error", async () => {
    const [
        deployer,
        -,
        --,
        vestingAccount,
        ...otherAccounts
    ] = await hre.ethers.getSigners();

    const GreenMintingToken = await
hre.ethers.getContractFactory("GreenMintingToken");
    const VestedLock = await
hre.ethers.getContractFactory("VestedLock");

    const vestedAmount = BigInt(90_000);
    const claimingPercentsSchedule = [3000, 2000, 5000]; // total =
100%
    const secPerStage = 10;

    const tokenHolderA = otherAccounts[0];
    const tokenHolderB = otherAccounts[1];

    const greenMintingToken = await GreenMintingToken.deploy(
        [tokenHolderA.address, tokenHolderB.address],
        [BigInt(0), BigInt(0)],
        vestedAmount
    );

    const currentBlock = await ethers.provider.getBlock("latest");
    const unvestStartTimestamp = currentBlock!.timestamp;

    const vestedLock = await VestedLock.deploy(
        vestingAccount.address,
        secPerStage,
        claimingPercentsSchedule,
        unvestStartTimestamp,
        greenMintingToken
    );

    await greenMintingToken.transfer(await vestedLock.getAddress(),
vestedAmount);

    console.log("\n Off-by-One PoC: Immediate Unlock Check");
    console.log(`    Current block timestamp:
${currentBlock.timestamp}`);
    console.log(`    Unvesting start timestamp:
${unvestStartTimestamp}`);
    console.log(`    Time passed:
${currentBlock.timestamp - unvestStartTimestamp}s`);

    const available = await vestedLock.availableVestedTokens();
    const expectedFirstStageUnlock = (vestedAmount *
BigInt(claimingPercentsSchedule[0])) / BigInt(10_000);

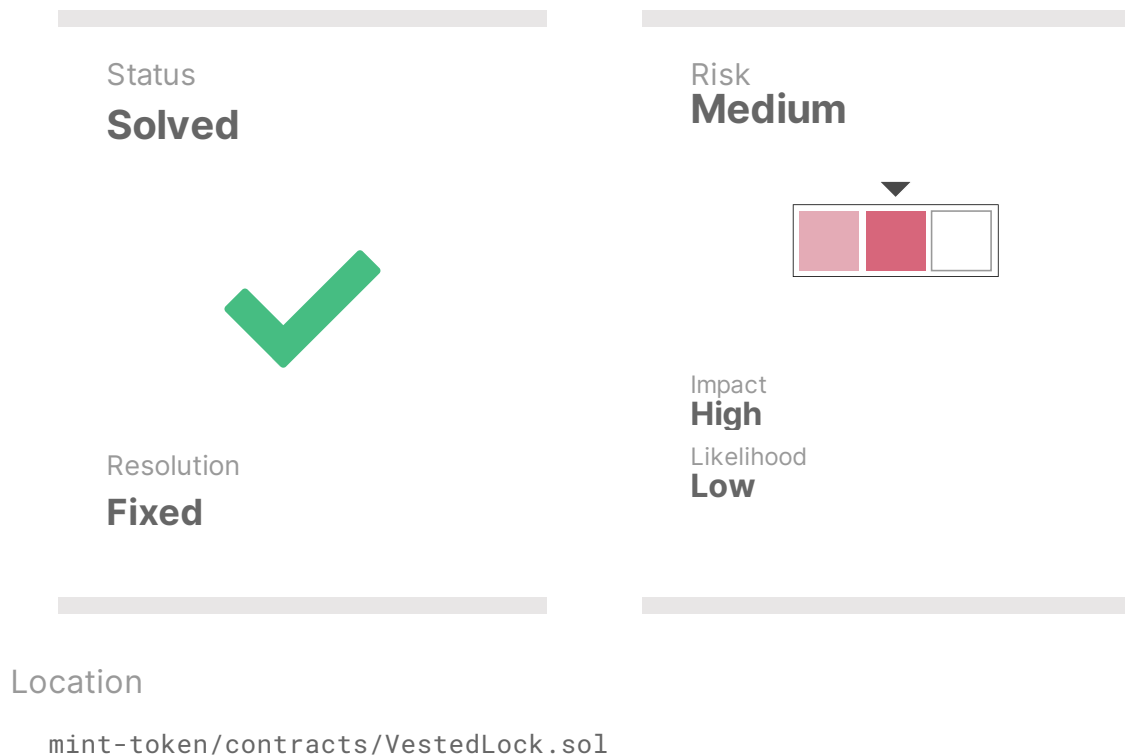
    console.log(`    Expected unlocked tokens (should be 0): 0`);
    console.log(`    ! Actual unlocked tokens:
${available}`);
    console.log(`    If actual > 0, the first vesting stage is

```

```
incorrectly active immediately.`);  
    expect(available).to.equal(0n);  
});
```


GRM-04

Missing constructor validation can lead to funds lock and vesting errors



Description

The constructor fails to validate critical parameters, allowing deployment of contracts with invalid vesting schedules that can result in permanent fund locks or incorrect vesting behavior.

The constructor accepts the `_claimingPercentsSchedule` array without performing essential validations.

It does not check if `_claimingPercentsSchedule` is empty. An empty schedule would cause immediate array out-of-bounds errors when `availableVestedTokens()` is called, as the function will always calculate `stage >= 1` but attempt to access non-existent array elements.

It does not verify that the sum of all percentages in `_claimingPercentsSchedule` equals `DENOMINATOR` (10000), which represents 100%. This can lead to two

problematic scenarios: - Under-allocation: if the sum is less than 10000, users will never be able to claim their full token allocation, leaving funds permanently locked in the contract - Over-allocation: if the sum exceeds 10000, the contract may attempt to distribute more than 100% of the tokens, causing it to revert due to insufficient balance

Recommendation

Add proper validation in the constructor to check the `_claimingPercentsSchedule` array is not empty and the sum of all percentages equals exactly `DENOMINATOR` (10,000).

Status

- Partially fixed on commit `201dc90a34c6cba40802c3b02f18d308047233b6`.

Added validation to ensure `claimingPercentsSchedule` is not empty. However, validation to ensure percentages sum to exactly 10000 (100%) was not implemented. Manual verification of the percentage sum is advised during deployment.

- Fixed on commit `0db99a2139e1e2a9d4b43ee9d66c7bc8501046fb`.

GRM-05

Missing zero address checks in constructor can break contract functionality

Status

Solved



Resolution

Fixed

Risk

None



Impact

Recommendation

Likelihood

—

Location

`mint-token/contracts/VestedLock.sol`

Description

Missing zero address checks in the constructor can result in a deployed contract with broken functionality that cannot be recovered without redeployment.

The constructor accepts `_vestingAccount` and `tokenAddress` parameters without validating they are not zero addresses. If `_vestingAccount` is set to `address(0)` during deployment, the `claimVestedTokens()` function will permanently fail since `msg.sender == vestingAccount` will never be true for `address(0)`, making all vested tokens permanently inaccessible. If `tokenAddress` is set to `address(0)` all token operations will fail, leaving the contract completely non-functional as it cannot interact with the ERC20 token.

Recommendation



Add zero address validation in the constructor for both `_vestingAccount` and `tokenAddress`.

Status

Fixed on commit `7db8644fb463abd435b78645ee163c3ce2f9a626`.

GRM-06

Unchecked token transfer can lead to silent failures and accounting inconsistencies

<div>Status</div> <div>Solved</div> <div></div> <div>Resolution</div> <div>Fixed</div>	<div>Risk</div> <div>None</div> <div></div> <div>Impact</div> <div>Recommendation</div> <div>Likelihood</div> <div>-</div>
<div>Location</div> <div><code>mint-token/contracts/VestedLock.sol</code></div>	

Description

The `claimVestedTokens()` function performs an unchecked ERC20 transfer that may cause issues with non-standard ERC20 tokens that return false on failure instead of reverting.

While the current token implementation reverts on transfer failure, if the contract is deployed with a token that does not revert, failed transfers would result in accounting mismatch where the contract's internal state is updated as if the transfer succeeded, but no tokens are actually transferred. This leads to reduced claimable amounts in future calculations based on the incorrectly updated `claimedVestedAmount`, and silent failure with no indication that the transfer failed.

Recommendation

Use OpenZeppelin's `SafeERC20` library or check the return value of the transfer function to ensure transfers succeed.

Status

Fixed on commit `7db8644fb463abd435b78645ee163c3ce2f9a626`.

Added OpenZeppelin's `SafeERC20` library and replaced the unchecked `token.transfer()` with `token.safeTransfer()`.

6. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.