Smart Contract Security Audit Report

For

TokenVesting

Date Issued: December.2, 2025

Version: v1.0

Confidentiality Level: Public

Contents

1 Abstract

This report was prepared for TokenVesting smart contract to identify issues and vulnerabilities in its smart contract source code. A thorough examination of TokenVesting smart contracts was conducted through timely communication with TokenVesting, static analysis using multiple audit tools and manual auditing of their smart contract source code.

The audit process paid particular attention to the following considerations.

- A thorough review of the smart contract logic flow
- Assessment of the code base to ensure compliance with current best practice and industry standards
- Ensured the contract logic met the client's specifications and intent
- Internal vulnerability scanning tools tested for common risks and writing errors
- Testing smart contracts for common attack vectors
- Test smart contracts for known vulnerability risks
- Conduct a thorough line-by-line manual review of the entire code base

As a result of the security assessment, issues ranging from critical to informational were identified. We recommend that these issues are addressed to ensure a high level of security standards and industry practice. The recommendations we made could have better served the project from a security perspective.

- Enhance general coding practices to improve the structure of the source code.
- Provide more comments for each function to improve readability.
- Provide more transparency of privileged activities once the agreement is in place.

2 Overview

2.1 Project Summary

Project Summary	Project Information
Name	TokenVesting
Start date	December 1, 2025
End date	December 2, 2025
Contract type	Vesting Contract
Language	Solidity

2.2 Report HASH

Name	HASH
TokenVesting	https://bscscan.com/address/0x6ec59ee782b8f8b2ce8c7360e189d3478633a48a

3 Project contract details

3.1 Contract Overview

TokenVesting.sol

The TokenVesting contract is an upgradeable ERC20 token linear release/allocation contract. Its function is for the project team to set the tokens, define different categories of beneficiaries (such as Seed, Strategic, Ecosystem, Team, Mining), and release them linearly over time according to the unlocking rules configured for each category (different total amount, different start month, different end month). Once the TGE start time is set, beneficiaries can call `claim()` to claim any unclaimed unlocked tokens when they meet the eligibility conditions. The contract also provides mechanisms such as Pausable, ReentrancyGuard, and Upgradeable Tokens (UUPS).

4 Audit results

4.1 Key messages

ID	Title	Severity	Status
01	Allocation Type uses a shared quota pool where multiple addresses share the same total allocation	Medium	Ignore
02	Owner Privileged Role	Low	Ignore
03	The `endMonths = 37` setting for `MINING` is inconsistent with the comment	Information	Ignore

4.2 Audit details

4.2.1 Allocation Type uses a shared quota pool where multiple addresses share the same total allocation.

ID	Severity	Location	Status
1	Medium	TokenVesting.sol	Ignore

Description

The contract sets a fixed total allocation for each of the five allocation types (Seed, Strategic, Ecosystem, Team, Mining):

uint256[5] memory totalAllocations = [60M, 60M, 78M, 90M, 778.05M];

However, the contract does not store "how many tokens this address should receive" for each beneficiary anywhere. The allocationType also does not restrict whether it can be shared by multiple people. The total allocation corresponding to the allocationType has no remaining quota record.

This results in all users of the same type sharing the same linear release curve, that is:

All addresses under the same allocationType will share the same total quota pool!

For example: Set 100 addresses to total allocation = 60,000,000 tokens. Each person's vested amount is calculated as:

return totalAlloc * timeElapsed / totalDuration;

Here, totalAlloc is the entire 60,000,000, not the quota due to a single user.

The result is that the first user to claim the prize received all 60 million, and subsequent users were no longer eligible to receive any.

Code location:

```
58      function setBeneficiary(address _beneficiary, uint256 _allocationType) external onlyOwner {
59          require(_allocationType <= 4, "Invalid allocation type");
60          require(_beneficiary != address(0), "Invalid address");
61
62          beneficiaryAllocationType[_beneficiary] = _allocationType + 1;
63      }
64
65
66      function computeVestedAmount(address _beneficiary) public view returns (uint256) {
67          uint256 allocType = beneficiaryAllocationType[_beneficiary];
68          if (allocType == 0) return 0;
69          allocType = allocType - 1;
70
71          if (tgeTime == 0 || block.timestamp < tgeTime) return 0;
72
73          uint256[5] memory totalAllocations = [
74              uint256(60_000_000 ether),    // 0: SEED_ROUND - 5%
75              uint256(60_000_000 ether),    // 1: STRATEGIC_ROUND - 5%
76              uint256(78_000_000 ether),    // 2: ECOSYSTEM - 10% * 65% (excluding 35% TGE)
77              uint256(90_000_000 ether),    // 3: TEAM - 7.5%
78              uint256(778_050_000 ether)    // 4: MINING - 65% * 99.75% (excluding 0.25% TGE)
79          ];
80
81          uint256[5] memory startMonths = [
82              uint256(12),  // 0: SEED_ROUND - start after 12 months
83              uint256(12),  // 1: STRATEGIC_ROUND - start after 12 months
84              uint256(0),   // 2: ECOSYSTEM - start immediately
85              uint256(12),  // 3: TEAM - start after 12 months
86              uint256(1)    // 4: MINING - start immediately
87          ];
88
89          uint256[5] memory endMonths = [
90              uint256(36),  // 0: SEED_ROUND - end at 36 months (12+24)
91              uint256(36),  // 1: STRATEGIC_ROUND - end at 36 months (12+24)
92              uint256(3),   // 2: ECOSYSTEM - end at 3 months (12 weeks)
93              uint256(36),  // 3: TEAM - end at 36 months (12+24)
94              uint256(37)   // 4: MINING - end at 36 months
95          ];
96
97          uint256 totalAlloc = totalAllocations[allocType];
98          uint256 startTime = tgeTime + startMonths[allocType] * MONTH;
99          uint256 endTime = tgeTime + endMonths[allocType] * MONTH;
100
101         if (block.timestamp < startTime) {
102             return 0;
103         }
104
105         if (block.timestamp >= endTime) {
106             return totalAlloc;
107         }
108
109         uint256 timeElapsed = block.timestamp - startTime;
110         uint256 totalDuration = endTime - startTime;
111
112         return totalAlloc * timeElapsed / totalDuration;
113     }
```

Recommendation

Record the individual allocated quota for each user and add a remaining quota check to the allocationType.

Status

Ignore.

4.2.2 Owner Privileged Role

ID	Severity	Location	Status
2	Low	TokenVesting.sol	Ignore

Description

The contract's core logic heavily relies on the `onlyOwner` permission. If the Owner's private key is leaked, stolen, or maliciously manipulated, the following risks may occur:

The Owner can arbitrarily modify user ownership relationships.

The Owner can call `setBeneficiary(address, uint256 allocationType)` at any time.

The Owner can remove an address from vesting (setting it to 0) at any time.

The Owner can change the `allocationType` to a higher amount.

The Owner can set its own address to the type with the highest reward.

The Owner can modify the vesting start time `startVesting()`.

The Owner can set a very distant future time, preventing users from ever claiming rewards.

The Owner can also set a very early time (even `block.timestamp`), skipping the cliff logic.

Recommendation

It is recommended to use multi-signature or Timelock to manage the Owner and restrict what the Owner can modify.

Status

Ignore.

4.2.3 The `endMonths = 37` setting for `MINING` is inconsistent with the comment.

ID	Severity	Location	Status
3	Information	TokenVesting.sol	Ignore

Description

In computeVestedAmount, the comment for MINING (allocationType = 4) is:

// 4: MINING – end at 36 months

However, the actual code is set as: uint256[5] memory endMonths = [

36,36,3,36,

37 // MINING

];

That is, the actual end month of MINING is 37 months, but the comment says 36 months.

Code implementation != Product requirements

Comment logic != Actual logic

Future maintenance may mistakenly assume the end time is 36 months.

Recommendation

It is recommended to correct the code or comments.

Status

Ignore.

5 Finding Categories

Centralization / Privilege

Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a struct assignment operation affecting an in-memory struct rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as constant contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Disclaimer

This report is issued in response to facts that occurred or existed prior to the issuance of this report, and liability is assumed only on that basis. Shield Security cannot determine the security status of this program and assumes no responsibility for facts occurring or existing after the date of this report. The security audit analysis and other content in this report is based on documents and materials provided to Shield Security by the information provider through the date of the insurance report. in Shield Security's opinion. The information provided is not missing, falsified, deleted or concealed. If the information provided is missing, altered, deleted, concealed or not in accordance with the actual circumstances, Shield Security shall not be liable for any loss or adverse effect resulting therefrom. shield Security will only carry out the agreed security audit of the security status of the project and issue this report. shield Security is not responsible for the background and other circumstances of the project. Shield Security is not responsible for the background and other circumstances of the project.