Alberich Token (ALBRH)

Smart Contract Security Audit Report

Conducted by **Hacken**

Audit completed: **June 8, 2026**

Audit firm: **Hacken**

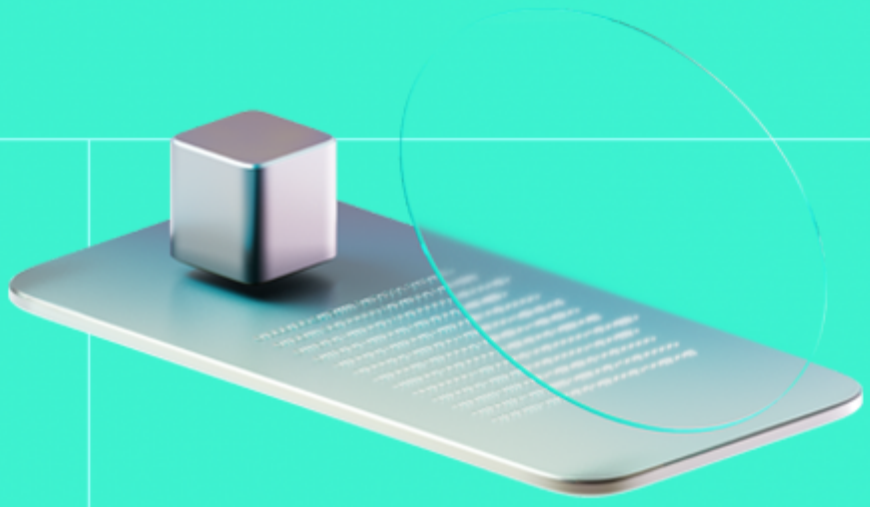
AUDIT SUMMARY

CRITICAL FINDINGS	HIGH FINDINGS	FINAL STATUS
0	0	AUDIT COMPLETED

This audit report reflects the security assessment of the Alberich Token (ALBRH) smart contract and its associated functionalities as of **June 8, 2026**.

HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Alberichtoken

Date: 08/06/2026

We express our gratitude to the Alberichtoken team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

AlberichToken is an ERC-20 token with extended functionality for token sales, vesting, snapshots, pausability, and reentrancy protection.

Document

Name	Smart Contract Code Review and Security Analysis Report for Alberichtoken
Audited By	Ataberk Yavuzer, Khrystyna Tkachuk
Approved By	Ivan Bondar
Website	N/A
Changelog	13/11/2025 - Preliminary Report
	26/11/2025 - Final Report
	04/06/2026 - Additional Review Preliminary Report
	08/06/2026 - Additional Review Final Report
Platform	EVM
Language	Solidity
Tags	ERC-20, Vesting, Token Sales
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review

Scope

File Name	Alberich-Token_Internal_Test_Results_Final_2025-10-29.zip
File Hash (SHA256)	98c4c5fb3d0477249ca8ba4fcffa275c17a9f565d175ec5b76e8c2127d792c3f
Final File Name	UPDATED_Alberich_Token_Audit_Ready_Smart_Contract_With_Fixes_2025-11-14.sol
Final File Hash (SHA256)	a26e0a4d05b91f5fa2bee218f024c61df4f3b6189df1976157898cf674b75bb4

Review

Scope

Additional Review File Name	UPDATED_Alberich_Token_Audit_Ready_Contract_With_Allocation_Fix_2026-05-27.sol
Additional Review File Hash (SHA256)	d395b4d0f052e1ae299021cd44ae7f4a5d12ee6d7c43e635116c53bb46ea921a
Additional Review Final File Name	UPDATED_Alberich_Token_Audit_Ready_Contract_Remediation_2026-06-04.sol
Additional Review FInal File Hash (SHA256)	d935eec51449f8890722adcba7f2438b91ee1b2fceed2459c437acdc7b9427d7

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

15	12	2	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	6
Low	3

Vulnerability	Severity	Status
F-2025-13933 - Unusable Emergency Pause Functionality	Medium	Fixed
F-2025-13936 - Freeze Mechanism Blocks Multi-Round Purchase Execution Flow	Medium	Fixed
F-2025-13952 - Unrestricted Burning of Reserved Tokens May Deplete Vesting Reserves	Medium	Fixed
F-2025-13953 - Insufficient Reserve Validation in Vesting Creation Cause Over-Allocation Risk	Medium	Fixed
F-2025-13968 - Single Vesting Limitation Prevents Creation of New Vesting for the Same Address After Completion of Existing One	Medium	Fixed
F-2026-17747 - Missing Vesting Reservation Floor in _purchase Leads to Permanently Insolvent Vesting Claims	Medium	Fixed
F-2025-13944 - Immediate Vesting Release Due to Missing Start Time Validation	Low	Fixed
F-2025-13965 - Missing Snapshot Function Can Prevent Core Functionality	Low	Fixed
F-2026-17748 - Irreversible Vesting Schedules With Unbounded Duration Can Permanently Lock Reserved Supply	Low	Accepted
F-2025-13934 - Floating Pragma	Info	Fixed
F-2025-13964 - Lack of Event Emitting for Key State Changes	Info	Fixed
F-2025-13966 - Lack of Two Step Ownership Pattern	Info	Mitigated
F-2025-13967 - Unbounded Loop in setWhitelist Allows Gas Griefing	Info	Fixed
F-2025-13969 - Staking Placeholder Functions Are Incomplete and Confusing	Info	Accepted

Vulnerability	Severity	Status
F-2026-17751 - Lack of Dedicated Event for Foundation Allocation Transfers	Info	Fixed

Documentation quality

- Functional requirements are provided.
- Technical description is provided.
  - Run instructions are provided.
  - Technical specification is provided.
- Inline comments exist partially throughout the codebase.

Code quality

- The code leverages OpenZeppelin and mostly follows best practices and style guides.
  - See informational findings for more details.
- The development environment is configured.

Test coverage

Code coverage of the project is **0%** since no tests were provided.

Table of Contents

System Overview

AlberichToken — An ERC-20 token with extended functionality for token sales, vesting, snapshots, pausability, and reentrancy protection. All tokens are minted to the contract itself at deployment, and distributed through controlled mechanisms (purchases, vesting, treasury). No additional minting is possible after deployment.

It has the following attributes:

- **Name:** Alberich Token
- **Symbol:** ALBRH
- **Decimals:** 18
- **Total Supply:** 87,600,000,000 tokens (87.6 billion ALBRH)
- **Initial Distribution:** 100% minted to contract address at deployment

Privileged roles

The `foundation` (must also hold `FOUNDATION_ROLE`): Controls all administrative, treasury, vesting, and lifecycle operations.

- Can call `setTokenPriceWeiPerToken` to set the token sale price in wei per token (blocked after parameter freeze).
- Can call `setRoundType` to switch between SHO and Public sale rounds (permitted even after parameter freeze).
- Can call `setPurchaseLimits` to configure per-wallet purchase caps for SHO and Public rounds (blocked after parameter freeze).
- Can call `setMinTokensPerPurchase` to set or disable the minimum token purchase threshold (blocked after parameter freeze).
- Can call `setWhitelist` to add or remove addresses from the SHO whitelist in batches (blocked after parameter freeze).
- Can call `setSaleActive` to activate or deactivate the token sale.
- Can call `freezeParameters` to permanently and irreversibly lock sale parameters (price, limits, whitelist, minimum purchase).
- Can call `pause` to pause all token transfers, purchases, and vesting claims.
- Can call `unpause` to resume all token transfers, purchases, and vesting claims.
- Can call `withdrawETH` to withdraw any amount of ETH from the contract to a specified address.
- Can call `rescueERC20` to transfer any ERC-20 token (excluding ALBRH itself) held by the contract to a specified address.
- Can call `ringForgedBurn` to permanently burn unlocked (non-vesting-reserved) ALBRH tokens held by the contract.
- Can call `setStakingContract` to set or update the staking contract address.
- Can call `toggleStakingBonusRequirement` to enable or disable the staking bonus requirement flag.
- Can call `setVesting` to create a linear vesting schedule for any beneficiary, reserving tokens from the contract balance.
- Can call `initiateFoundationTransfer` to nominate a new foundation address, initiating a two-step governance handoff.

- Can call `snapshot` to create a new ERC-20 balance snapshot.
- Can call `transferAllocation` to transfer unlocked (non-vesting-reserved) ALBRH tokens from the contract to any address.

The holder of the `DEFAULT_ADMIN_ROLE` can:

- Grant or revoke any role, including `FOUNDATION_ROLE` and `DEFAULT_ADMIN_ROLE`
- Acts as a super-administrator over the entire role hierarchy

The `pendingFoundation`: Address nominated via `initiateFoundationTransfer` to receive governance control.

- Can call `acceptFoundationRole` to complete the two-step handoff, becoming the new `foundation` address, receiving `FOUNDATION_ROLE`, and revoking `FOUNDATION_ROLE` from the previous foundation.

Potential Risks

- **System Reliance on External Contracts:** The functioning of the system significantly relies on specific external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
  - Applies to: Future staking contract integration (stakingContract address)
- **Fixed Total Supply Post-Deployment:** The token's total supply is fixed at deployment and cannot be adjusted afterward, which may limit the project's adaptability and flexibility in managing its economic model.
  - Applies to: 87.6B ALBRH tokens minted at deployment
- **Centralized Minting to a Single Address:** The project concentrates minting tokens in a single address, raising the risk of fund mismanagement or theft, especially if key storage security is compromised.
  - Applies to: All tokens minted to contract address at deployment
- **Dynamic Array Iteration Gas Limit Risks:** The project iterates over large dynamic arrays, which leads to excessive gas costs, risking denial of service due to out-of-gas errors, directly impacting contract usability and reliability.
  - Applies to: setWhitelist() function
- **Owner's Unrestricted State Modification:** The absence of restrictions on state variable modifications by the owner leads to arbitrary changes, affecting contract integrity and user trust, especially during critical operations like minting phases.
  - Applies to: `FOUNDATION_ROLE` can modify all parameters before `freezeParameters()`
- **Coarse-grained Authorization Model Risks:** The broad authorization model increases the risk of protocol control loss if any authorized address is compromised, potentially leading to unauthorized actions and significant financial loss.
  - Applies to: Single `FOUNDATION_ROLE` controls all critical functions
- **Absence of Time-lock Mechanisms for Critical Operations:** Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions, increasing the risk of rapid exploitation and irreversible changes.
  - Applies to: All admin functions execute immediately (parameter changes, governance handoff, treasury operations)
- **Incomplete Role Transfer During Foundation Handoff**: The `acceptFoundationRole` function transfers `FOUNDATION_ROLE` to the new address and revokes it from the previous holder, but does not address `DEFAULT_ADMIN_ROLE`, which is granted to the deployer in the constructor via `_grantRole(DEFAULT_ADMIN_ROLE, msg.sender)`. Under OpenZeppelin's `AccessControl` model, the `DEFAULT_ADMIN_ROLE` holder retains the ability to call `grantRole` and `revokeRole` for any role, including `FOUNDATION_ROLE`. If the original deployer does not explicitly renounce `DEFAULT_ADMIN_ROLE` after handoff, they retain the capability to revoke `FOUNDATION_ROLE` from the new foundation or to grant it to an arbitrary address which may brick the execution of functions gated by the `onlyFoundation` modifier.

Findings

Vulnerability Details

F-2025-13933 - Unusable Emergency Pause Functionality - Medium

Description:

The `AlberichToken` contract implements the `ERC20` token standard with embedded functionality for managing Alberich Token (`ALBRH`) sales. It inherits OpenZeppelin's `Pausable` module, which provides an internal framework for emergency stops. The `AlberichToken` contract correctly applies the `whenNotPaused` modifier to critical functions such as `buy()` and `claimVestedTokens()` to restrict their execution during paused states.

The `AlberichToken.sol` contract includes the necessary components for a pausable system:

- It inherits `Pausable`:

```
contract AlberichToken is ERC20, ERC20Snapshot, AccessControl, Pausable, Reen
trancyGuard {
```

- It uses the `whenNotPaused` modifier on critical functions:

```
...
external
payable
nonReentrant
whenNotPaused
...
```

However, the contract fails to expose any public or external interface to trigger the pause mechanism. The `Pausable` module only defines internal `_pause()` and `_unpause()` functions, which must be explicitly wrapped by the inheriting contract with proper access control. Since `AlberichToken` does not implement such wrappers, the emergency stop mechanism remains inaccessible.

The lack of any public entry point that triggers internal `_pause()` and `_unpause()` renders the emergency control system ineffective. In the event of a detected exploit, logic malfunction, or unforeseen vulnerability, there would be no way to halt contract operations on-chain. This significantly increases systemic risk, as administrators cannot stop token transfers, purchases, or vesting claims during critical incidents.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Expose properly access-controlled public functions (e.g., `pause()` and `unpause()`) that internally invoke `_pause()` and `_unpause()`.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the external functions were added that expose the pausable functionality:

```
function pause() external onlyFoundation {
    _pause();
}

function unpause() external onlyFoundation {
    _unpause();
}
```

F-2025-13936 - Freeze Mechanism Blocks Multi-Round Purchase Execution Flow - Medium

Description:

The `AlberichToken` contract exhibits a design inconsistency between its round management logic and the parameter freeze mechanism.

The `AlberichToken` defines a multi-round architecture that differentiates between SHO and Public sale phases, maintaining independent purchase tracking (`purchasedSHO` and `purchasedPublic`) and corresponding limits (`shoPurchaseLimit` and `publicPurchaseLimit`). This structure implies a sequential sale process, where participants first engage in an SHO round and later transition to a Public round.

```
function _purchase() internal {
   ...

    if (isSHORound) {
        require(whitelist[msg.sender], "not whitelisted");
    }

    if (isSHORound) {
        uint256 newTotal = purchasedSHO[msg.sender] + tokensOut;
        require(newTotal <= shoPurchaseLimit, "exceeds SHO cap");
        purchasedSHO[msg.sender] = newTotal;
    } else {
        uint256 newTotal = purchasedPublic[msg.sender] + tokensOut;
        require(newTotal <= publicPurchaseLimit, "exceeds public cap");
        purchasedPublic[msg.sender] = newTotal;
    }

   ...
}
```

However, once parameters are frozen through the `freezeParameters()` function, the system enters a locked state that prevents any further modification of the round type. The `setRoundType()` function, responsible for toggling between sale phases, can only be called when the parameters remain unfrozen. Because purchases require the frozen state (`parametersFrozen == true`), the contract effectively becomes trapped in whichever round was active at the time of freezing. This contradiction renders the second sale phase unreachable and defeats the intended purpose of having a multi-round purchase mechanism.

```
function setRoundType(bool sho) external onlyFoundation {
    require(!parametersFrozen, "params frozen");
```

```
    isSHORound = sho;
    emit RoundSwitched(sho);
}

function freezeParameters() external onlyFoundation {
    require(!parametersFrozen, "already frozen");
    require(tokenPriceWeiPerToken > 0, "price unset");
    parametersFrozen = true;
    emit ParametersFrozen();
}

function _purchase() internal {
    require(parametersFrozen, "params not frozen");
   ...
}
```

The current design causes a complete breakdown of the intended multi-round sale workflow. After parameters are frozen, the contract can only facilitate transactions within a single round—either SHO or Public—while the other remains permanently disabled. This state dependency eliminates operational flexibility and prevents executing consecutive sale phases under a single deployment.

As a result, the implemented multi-round logic, data structures, and purchase tracking mechanisms serve no functional purpose. The inability to conduct both rounds as planned may lead to investor confusion, disrupt the token distribution strategy, restricted participation, and undermine the reliability of the sale process. In practical terms, the system's architecture supports two sale phases, but its operational logic enforces only one, creating a direct contradiction between the contract's design intent and its actual runtime behavior.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider allowing controlled round switching after parameters are frozen by removing the restriction in `setRoundType()`. This adjustment maintains the immutability of core parameters while enabling sequential execution of SHO and Public sale rounds. Alternatively, if the logic does not require multiple sale phases, simplify the design by removing redundant variables and logic related to round management to prevent misinterpretation and reduce contract complexity.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the `setRoundType()` function allows controlled round switching after parameters are frozen:

```
function setRoundType(bool sho) external onlyFoundation {
    // Allow switching even after parameters are frozen (for SHO -> Public se
quencing)
    isSHORound = sho;
    emit RoundSwitched(sho);
}
```

F-2025-13952 - Unrestricted Burning of Reserved Tokens May Deplete Vesting Reserves - Medium

Description:

The `AlberichToken` contract implements a `ringForgedBurn()` function, allowing the `FOUNDATION_ROLE` to burn tokens held by the contract's own address. However, there is no validation to ensure that the tokens being burned are not part of the amounts reserved to fulfill existing vesting schedules.

Specifically, the contract does not maintain a global state variable to track or reserve the total amount of tokens locked for vesting. As a result, invoking `ringForgedBurn()` may destroy tokens that are expected to be distributed to users under active vesting agreements. Once these tokens are burned, the contract will no longer have sufficient balance to honor vesting claims, resulting in failed withdrawals and permanent loss of user entitlements.

```
function ringForgedBurn(uint256 amount) external onlyFoundation whenNotPaused
{
    require(amount > 0, "zero");
    require(balanceOf(address(this)) >= amount, "insufficient");
    _burn(address(this), amount);
    emit RingForgedBurnExecuted(amount);
}
```

If the foundation executes `ringForgedBurn()` without considering the tokens reserved for active vestings, users may lose access to their vested tokens. This effectively breaks the vesting mechanism, causes denial of service for legitimate claims, and results in an unrecoverable supply inconsistency between total vesting commitments and the contract's token balance.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 5/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider introducing a reserve tracking mechanism that records the total amount of tokens locked for active vestings (e.g., `reservedForVesting` state variable). Enforce a validation check in `ringForgedBurn()` to ensure the burned amount does not exceed the reserved vesting balance.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), validation check was added in `ringForgedBurn()` to check if the burned amount does not exceed the reserved vesting balance:

```
function ringForgedBurn(uint256 amount) external onlyFoundation whenNotPaused
{
    require(amount > 0, "zero");

    uint256 contractBalance = balanceOf(address(this));
    require(contractBalance >= amount, "insufficient");

    // Prevent burning tokens that are reserved for vesting
    uint256 unlockedBalance = contractBalance - reservedForVesting;
    require(amount <= unlockedBalance, "insufficient unlocked");

    _burn(address(this), amount);
    emit RingForgedBurnExecuted(amount);
}
```

Evidences

PoC

Reproduce: The `BurnVestingReserve_PoC.t.sol` was introduced to demonstrate the issue:

- Retrieve the contract's total token balance and confirm it equals `TOTAL_SUPPLY`.
- Define vesting amount as `TOTAL_SUPPLY` (maximum possible value).
- Create Vesting for `beneficiary1` with amount = `TOTAL_SUPPLY`.
- Calling burn function to burn `TOTAL_SUPPLY` token, balance is zero.
- The total reserved amount (`TOTAL_SUPPLY`) now exceeds the contract's actual balance (`0`).
- Beneficiary 1 failed to claim — contract has insufficient balance to cover the legitimate vesting.

The PoC demonstrates that the contract allows burning tokens that are already reserved for vesting, leading to a depletion of the token pool required to fulfill active vesting obligations and potential claim failures for beneficiaries.

PoC code:

```
function test_BurnVestingReservedTokens_PoC() public {
    uint256 contractBalance = token.balanceOf(address(token));
    assertEq(contractBalance, TOTAL_SUPPLY);

    // vestingAmount for each vesting = TOTAL_SUPPLY, the max available value
    uint256 vestingAmount = TOTAL_SUPPLY;

    console.log("Contract balance:", contractBalance);
    console.log("Each vesting amount:", vestingAmount);

    // Create Vesting 1
    vm.prank(foundation);
    token.setVesting(
        beneficiary1,
        vestingAmount,
        uint64(block.timestamp),
        100 days
    );

    // Verify contract still has same balance (tokens not transferred yet)
    assertEq(token.balanceOf(address(token)), contractBalance);

    // Issue: burning the reserved token for vesting
    vm.prank(foundation);
    token.ringForgedBurn(TOTAL_SUPPLY);

    // Contract balance unchanged
    assertEq(token.balanceOf(address(token)), 0);

    // Calculate total reserved
    uint256 totalReserved = vestingAmount;
    console.log("Total reserved for vestings:", totalReserved);
    console.log("Available balance:", token.balanceOf(address(token)));

    // Issue: Now the contract has 0 balance but owes vestingAmount to benefi
ciary1

    vm.warp(block.timestamp + 200 days); // Fast forward to full vesting
    // Beneficiary 1 claims - SUCCEEDS
    vm.prank(beneficiary1);
    vm.expectRevert();
    token.claimVestedTokens(); // ← REVERTS: ERC20 transfer fails due to insu
```

```
fficient balance
}
```

Results:

```
Ran 1 test for test/BurnVestingReserve_PoC.t.sol:BurnVestingReserve_PoC
[PASS] test_BurnVestingReservedTokens_PoC() (gas: 132058)
Logs:
  Contract balance: 876000000000000000000000000
  Each vesting amount: 876000000000000000000000000
  Total reserved for vestings: 876000000000000000000000000
  Available balance: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.30ms (842.75µs
CPU time)
```

Files: BurnVestingReserve_PoC.t.sol

F-2025-13953 - Insufficient Reserve Validation in Vesting Creation Cause Over-Allocation Risk - Medium

Description:

When a new vesting is created, the contract only verifies that `balanceOf(address(this)) >= total` for the current vesting being initialized, where `total` point to the total allocated value for the current created vesting. However, it does not account for tokens already reserved for previously created vestings. As a result, multiple vestings can be created whose combined required amount exceeds the actual token balance held by the contract, leading to an overallocation of vesting liabilities.

```
function setVesting(address beneficiary, uint256 total, uint64 start, uint64
duration) external onlyFoundation {
    require(beneficiary != address(0), "zero");
    require(total > 0, "zero total");
    require(duration > 0, "zero dur");
    Vesting storage v = vestings[beneficiary];
    require(!v.exists, "exists");
    require(balanceOf(address(this)) >= total, "no inventory");
    vestings[beneficiary] = Vesting({total: total, released: 0, start: start,
duration: duration, exists: true});
}
```

This design flaw can result in the contract being unable to fulfill all vesting claims. Even though each individual vesting passes the initial balance check, the aggregate reserved amount may surpass the available balance. Consequently, when beneficiaries attempt to claim vested tokens, the contract may not hold enough tokens to satisfy legitimate entitlements. This can cause partial or total failure of token distribution, undermine user trust, and create potential legal and reputational exposure.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 5/5

Likelihood Rate: 4/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Consider implementing an aggregate reserve accounting mechanism to ensure that total outstanding vesting obligations never exceed the available token balance:

- Introduce a `reservedForVesting` variable to track the sum of all outstanding vesting amounts (`total - released`).
- Update `reservedForVesting` whenever a vesting is created or claimed.
- During vesting creation, validate that `balanceOf(address(this)) >= reservedForVesting + newTotal`.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the `reservedForVesting` variable was introduced to track the amount of tokens reserved for vesting, and appropriate validation was added in the `setVesting()` function to ensure that the contract's current balance is sufficient to cover both the newly created vesting and the previously committed vesting obligations.

```
function setVesting(
    address beneficiary,
    uint256 total,
    uint64 start,
    uint64 duration
) external onlyFoundation {
    ...
    uint256 contractBalance = balanceOf(address(this));
    // Ensure we do not over-allocate beyond contract balance
    require(contractBalance >= reservedForVesting + total, "no inventory");
    ...
    reservedForVesting += total;

    emit VestingCreated(beneficiary, total, start, duration);
}

function claimVestedTokens() external nonReentrant whenNotPaused {
    ...

    v.released += releasable;
    reservedForVesting -= releasable;

    ...
}
```

Evidences

PoC

Reproduce:

The `VestingOverallocation_PoC.t.sol` was introduced to demonstrate the issue:

- Retrieve the contract's total token balance and confirm it equals `TOTAL_SUPPLY`.
- Define each vesting amount as `TOTAL_SUPPLY` (maximum possible value).
- Create Vesting for `beneficiary1` with amount = `TOTAL_SUPPLY`.
- Create Vesting for `beneficiary2` with amount = `TOTAL_SUPPLY`.
- The total reserved amount (`vestingAmount * 2`) now exceeds the contract's actual balance.
- Beneficiary 1 claims successfully — contract transfers `TOTAL_SUPPLY`.
- Beneficiary 2 claim reverts — contract has insufficient balance to cover the second vesting.

The PoC demonstrate that the contract allows creation of multiple vestings whose combined reserved amounts exceed available tokens, leading to insolvency when multiple beneficiaries claim simultaneously.

PoC code:

```
function test_VestingOverAllocation_PoC() public {
    uint256 contractBalance = token.balanceOf(address(token));
    assertEq(contractBalance, TOTAL_SUPPLY);

    // vestingAmount for each vesting = TOTAL_SUPPLY, the max available value
    uint256 vestingAmount = TOTAL_SUPPLY;

    console.log("Contract balance:", contractBalance);
    console.log("Each vesting amount:", vestingAmount);
    console.log("Total vesting liabilities:", vestingAmount * 2);

    // Create Vesting 1
    vm.prank(foundation);
    token.setVesting(
        beneficiary1,
        vestingAmount,
        uint64(block.timestamp),
        100 days
    );
    console.log("Vesting 1 created for:", vestingAmount);

    // Verify contract still has same balance (tokens not transferred yet)
    assertEq(token.balanceOf(address(token)), contractBalance);
```

```
    // Create Vesting 2
    vm.prank(foundation);
    token.setVesting(
        beneficiary2,
        vestingAmount,
        uint64(block.timestamp),
        100 days
    );
    console.log("Vesting 2 created for:", vestingAmount);

    // Contract balance unchanged
    assertEq(token.balanceOf(address(token)), contractBalance);

    // Calculate total reserved
    uint256 totalReserved = vestingAmount * 2;
    console.log("Total reserved for vestings:", totalReserved);
    console.log("Available balance:", contractBalance);

    // Issue: Total reserved exceeds available balance
    // If beneficiaries claim simultaneously, contract will not have enough t
okens

    vm.warp(block.timestamp + 200 days); // Fast forward to full vesting
    // Beneficiary 1 claims - SUCCEEDS
    vm.prank(beneficiary1);
    token.claimVestedTokens();
    assertEq(token.balanceOf(beneficiary1), vestingAmount);
    // Beneficiary 2 claims - FAILS! Contract doesn't have enough tokens
    vm.prank(beneficiary2);
    vm.expectRevert(); // ERC20 transfer will revert due to insufficient bala
nce
    token.claimVestedTokens(); // ← REVERTS: ERC20 transfer fails due to insu
fficient balance
}
```

Results:

```
Ran 1 test for test/VestingOverallocation_PoC.t.sol:VestingOverallocation_PoC
[PASS] test_VestingOverAllocation_PoC() (gas: 231566)
Logs:
  Contract balance: 876000000000000000000000000
  Each vesting amount: 876000000000000000000000000
  Total vesting liabilities: 1752000000000000000000000000
  Vesting 1 created for: 876000000000000000000000000
  Vesting 2 created for: 876000000000000000000000000
  Total reserved for vestings: 1752000000000000000000000000
  Available balance: 876000000000000000000000000
```

```
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.79ms (1.13ms C
PU time)
```

Files: VestingOverallocation_PoC.t.sol

F-2025-13968 - Single Vesting Limitation Prevents Creation of New Vesting for the Same Address After Completion of Existing One - Medium

Description:

The vesting logic restricts each beneficiary to a single vesting schedule by enforcing `require(!v.exists, "exists")` during creation. The `exists` flag is set to `true` when a vesting is initialized and is never reset, even after the schedule is fully vested and all tokens have been released. This design creates a permanent lock on the beneficiary's address, preventing the creation of any subsequent vesting entries.

In the current implementation, the flag remains `true` indefinitely, as there is no condition in `claimVestedTokens()` or any cleanup function that clears or reinitializes the struct once the vesting is completed. Consequently, any future call to `setVesting()` for the same address will revert, regardless of whether the previous vesting has ended or all tokens have been claimed.

```
function setVesting(address beneficiary, uint256 total, uint64 start, uint64
duration) external onlyFoundation {
    require(beneficiary != address(0), "zero");
    require(total > 0, "zero total");
    require(duration > 0, "zero dur");
    Vesting storage v = vestings[beneficiary];
    require(!v.exists, "exists");
    require(balanceOf(address(this)) >= total, "no inventory");
    vestings[beneficiary] = Vesting({total: total, released: 0, start: start,
duration: duration, exists: true});
}
```

A beneficiary address becomes permanently restricted to one vesting schedule. After the initial vesting is created, it is impossible to assign new vesting parameters even if the previous schedule has concluded. This behavior prevents contract re-use for new distributions or adjustments and leads to rigid, non-reusable state.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate:	3/5
Likelihood Rate:	5/5
Exploitability:	Dependent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: Consider allowing reinitialization of vesting once the previous schedule is fully completed. This can be achieved by resetting the `exists` flag or deleting the vesting record when `released >= total`. Alternatively, implement indexed storage to support multiple independent vestings per address without overwriting or blocking state.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the vesting is deleted once the total amount was vested and claimed:

```
function claimVestedTokens() external nonReentrant whenNotPaused {
    ...

    // If vesting is fully completed, allow a new vesting in future
    if (v.released == v.total) {
        delete vestings[msg.sender];
    }

    _transfer(address(this), msg.sender, releasable);
    emit VestedTokensClaimed(msg.sender, releasable);
}
```

F-2026-17747 - Missing Vesting Reservation Floor in _purchase Leads to Permanently Insolvent Vesting Claims - Medium

Description:

In **AlberichToken**, the `_purchase()` function validates available inventory using only the contract's raw token balance:

```
require(balanceOf(address(this)) >= tokensOut, "insufficient");
_transfer(address(this), msg.sender, tokensOut);
```

The `reservedForVesting` state variable is not subtracted before this check. By contrast, `ringForgedBurn()` and `transferAllocation()` both compute an unlocked balance:

```
uint256 unlockedBalance = contractBalance - reservedForVesting;
require(amount <= unlockedBalance, "insufficient unlocked");
```

This asymmetry allows the sale path to drain the contract's token balance below the amount reserved for active vesting schedules. Once the balance is depleted below `reservedForVesting`, any subsequent call to `claimVestedTokens()` reverts inside ERC20 `_transfer()` because the contract holds fewer tokens than the releasable amount. Since no `mint()` function exists and sold tokens cannot be reclaimed, the shortfall is permanent until the contract balance is topped up with additional tokens explicitly via `transfer()` to the contract address. Beneficiaries with active vesting schedules lose their entire vested allocation with no direct recovery path. Routine sale traffic overlapping a live vesting schedule suffices to trigger insolvency.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 3/5

Likelihood Rate: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: Subtract the vesting reservation before validating inventory in `_purchase()`, mirroring the guard used by the other outflow paths:

```
uint256 unlockedBalance = balanceOf(address(this)) - reservedForVesting;
require(tokensOut <= unlockedBalance, "insufficient unlocked");
```

Resolution: **Fixed in `d935eec`.**

The `_purchase()` function in **AlberichToken** now computes an unlocked balance by subtracting `reservedForVesting` from the contract's token balance before validating purchase inventory. The updated validation check prevents sale outflows from encroaching on tokens reserved for active vesting schedules.

```
// Ensure we don't sell tokens reserved for vesting
uint256 unlockedBalance = balanceOf(address(this)) - reservedForVesting;
require(tokensOut <= unlockedBalance, "insufficient unlocked");
```

Evidences

_purchase Ignores reservedForVesting, Allowing Ordinary Purchases to Render Vesting Claims Permanently Insolvent

Reproduce: **PoC Steps:**

1. Foundation deploys **AlberichToken**, sets price, freezes params, switches to public round, activates sale
2. Foundation creates a vesting schedule of 1,000 tokens for `beneficiary`
3. Foundation uses `transferAllocation` to drain all unlocked tokens to a sink, leaving the contract balance at exactly `reservedForVesting` (1,000 tokens)
4. `buyer` calls `buy()` with ETH worth 1,000 tokens — `_purchase` only checks `balanceOf(this) >= tokensOut` and succeeds, draining balance to 0
5. Assert contract balance is 0 while `reservedForVesting` is still 1,000 tokens (invariant broken)
6. Warp past vesting end, confirm `vestedAmount == VESTING_AMOUNT` (fully vested)
7. `beneficiary` calls `claimVestedTokens()` — reverts because the contract has zero balance to transfer

PoC Code:

```
function test_purchaseDrainsVestingReserve() public {
    uint256 ethNeeded = (VESTING_AMOUNT * TOKEN_PRICE) / 1e18;
    vm.deal(buyer, ethNeeded);

    // _purchase checks only balanceOf(this) >= tokensOut, ignoring reservedF
orVesting
    vm.prank(buyer);
    token.buy{value: ethNeeded}();

    assertEq(token.balanceOf(address(token)), 0);
    assertEq(token.balanceOf(buyer), VESTING_AMOUNT);
    assertEq(token.reservedForVesting(), VESTING_AMOUNT, "reserve accounting
unchanged");

    // Warp past full vesting period
    vm.warp(block.timestamp + 366 days);

    uint256 claimable = token.vestedAmount(beneficiary);
    assertEq(claimable, VESTING_AMOUNT, "fully vested");

    vm.prank(beneficiary);
    vm.expectRevert();
    token.claimVestedTokens();
}
```

Results:

```
Ran 1 test for test/PoC_PurchaseIgnoresVestingReserve.t.sol:PoC_PurchaseIgnor
esVestingReserve
[PASS] test_purchaseDrainsVestingReserve() (gas: 125167)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.59ms (600.75µs
CPU time)
```

Files: PoC_PurchaseIgnoresVestingReserve.t.sol

F-2025-13944 - Immediate Vesting Release Due to Missing Start Time Validation - Low

Description:

The `AlberichToken` contract allows the creation of linear vesting schedules through the `setVesting()` function, which accepts a `start` timestamp parameter to determine when the vesting process begins. However, the contract does not validate whether the provided start time is in the past. As a result, a vesting schedule can be initialized with a past timestamp, causing the vesting calculation logic to consider the entire or partial duration as already elapsed. Consequently, users may immediately claim the full or a disproportionate portion of their vested tokens upon creation, bypassing the intended linear release schedule.

```
function setVesting(address beneficiary, uint256 total, uint64 start, uint64
duration) external onlyFoundation {
    require(beneficiary != address(0), "zero");
    require(total > 0, "zero total");
    require(duration > 0, "zero dur");
    Vesting storage v = vestings[beneficiary];
    require(!v.exists, "exists");
    require(balanceOf(address(this)) >= total, "no inventory");
    vestings[beneficiary] = Vesting({total: total, released: 0, start: start,
duration: duration, exists: true});
}
```

If the `start` timestamp is set to a past time (intentionally or accidentally), the vesting logic will treat the vesting period as already completed or partially completed. This may lead to an unexpected premature unlock of tokens, disrupting the token emission schedule and potentially impacting tokenomics or supply control mechanisms.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement a validation check within the `setVesting()` function to ensure that the provided `start` timestamp is not earlier than the current block timestamp. This ensures that vesting schedules behave predictably and tokens are released only over the intended duration.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), `start` timestamp is now validating to be not set in the past:

```
function setVesting(
    address beneficiary,
    uint256 total,
    uint64 start,
    uint64 duration
) external onlyFoundation {
    ...
    require(start >= block.timestamp, "start in past");
    ...
}
```

F-2025-13965 - Missing Snapshot Function Can Prevent Core Functionality - Low

Description:

The contract inherits from `ERC20Snapshot` but does not expose any function to create snapshots. The `_snapshot()` internal function is never called, making the snapshot functionality completely inaccessible.

AlberichToken.sol:

```
contract AlberichToken is ERC20, ERC20Snapshot, AccessControl, Pausable, Reen
trancyGuard {
    // ... no _snapshot() call anywhere

    function _beforeTokenTransfer(address from, address to, uint256 amount)
        internal
        override(ERC20, ERC20Snapshot)  // Overriding but not using snapshots
    {
        require(!paused(), "paused");
        super._beforeTokenTransfer(from, to, amount);
    }
}
```

Impact:

- If snapshots are intended for governance/voting, the feature is completely broken
- Gas waste from inheriting unused functionality
- Override of `_beforeTokenTransfer` adds complexity without benefit
- If snapshots are needed later, tokens already distributed cannot be snapshotted retroactively

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to utilize the `_snapshot()` functionality if it was designed for any use case. Unless, it is recommended to remove that inheritance from the code to prevent code clutter.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the `_snapshot()` functionality was implemented:

```
function snapshot() external onlyFoundation returns (uint256) {
    uint256 id = _snapshot();
    emit SnapshotCreated(id);
    return id;
}
```

F-2026-17748 - Irreversible Vesting Schedules With Unbounded Duration Can Permanently Lock Reserved Supply - Low

Description:

In **AlberichToken**, the `setVesting()` function creates vesting schedules that increment `reservedForVesting` by the total vested amount. The only path to decrement this variable is a successful call to `claimVestedTokens()` by the beneficiary.

```
function setVesting(
    address beneficiary,
    uint256 total,
    uint64 start,
    uint64 duration
) external onlyFoundation {
    require(beneficiary != address(0), "zero");
    require(total > 0, "zero total");
    require(duration > 0, "zero dur");
    require(start >= block.timestamp, "start in past");

    Vesting storage v = vestings[beneficiary];
    require(!v.exists, "exists");

    uint256 contractBalance = balanceOf(address(this));
    // Ensure we do not over-allocate beyond contract balance
    require(contractBalance >= reservedForVesting + total, "no inventory");

    vestings[beneficiary] = Vesting({
        total: total,
        released: 0,
        start: start,
        duration: duration,
        exists: true
    });

    reservedForVesting += total;

    emit VestingCreated(beneficiary, total, start, duration);
}
```

However, no cancel, modify, or reclaim function exists. The `start` and `duration` parameters are `uint64` values bounded only by `require(start >= block.timestamp)`, with no upper limit enforced. If a beneficiary loses access to their key, or beneficiary is a contract incapable of calling `claimVestedTokens()`, or if `start` is mistakenly set far in the future, the

reserved tokens remain locked indefinitely. The `reservedForVesting` variable stays permanently inflated, reducing the pool available to `ringForgedBurn()` and `transferAllocation()`. This represents an irrecoverable supply-availability degradation.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Accepted

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Implement a foundation-gated `cancelVesting` function that reclaims only the **unvested** portion, preserving the beneficiary's right to claim already-vested tokens. To prevent misuse as a rug vector, restrict full cancellation (including vested-but-unclaimed tokens) to a grace period after the vesting schedule has fully elapsed and the beneficiary has not claimed:

```
uint256 public constant CLAIM_GRACE_PERIOD = 180 days;

function cancelVesting(address beneficiary) external onlyFoundation {
    Vesting storage v = vestings[beneficiary];
    require(v.exists, "no vesting");

    uint256 vested = vestedAmount(beneficiary);
    uint256 unvested = v.total - vested;

    uint256 vestingEnd = uint256(v.start) + uint256(v.duration);
    bool graceElapsed = block.timestamp > vestingEnd + CLAIM_GRACE_PERIOD;

    if (graceElapsed) {
        // Beneficiary never claimed after full vesting + grace — reclaim all
unclaimed
        uint256 unclaimed = v.total - v.released;
        reservedForVesting -= unclaimed;
```

```
            delete vestings[beneficiary];
            emit VestingCancelled(beneficiary, unclaimed);
        } else {
            // Mid-vesting: only reclaim the unvested portion
            require(unvested > 0, "fully vested");
            reservedForVesting -= unvested;
            v.total = vested;
            emit VestingCancelled(beneficiary, unvested);
        }
    }
```

Additionally, enforce a reasonable upper bound on `start` and `duration` to prevent accidental far-future schedules.

Resolution: **Accepted.**

The Alberich team acknowledged the finding with the following context:

- Vesting schedules are limited to known founder allocations.
- Beneficiary wallets are secured using hardware wallets and recovery phrases.
- Vesting parameters are fixed and verified prior to deployment.
- The vesting structure is fully transparent and aligned with the published tokenomics.

F-2025-13934 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.20`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The `AlberichToken` contract currently use floating pragma declarations `^0.8.20`. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example,

instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.xx;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), the pragma version is locked `pragma solidity 0.8.20`.

F-2025-13964 - Lack of Event Emitting for Key State Changes - Info

Description:

The `AlberichToken` defines storage variables that hold key data, which can later be modified during contract execution within the appropriate setter functions:

Affected functions:

- `setTokenPriceWeiPerToken()`
- `setPurchaseLimits()`
- `setMinTokensPerPurchase()`
- `setWhitelist()`
- `rescueERC20()`
- `setVesting()`
- `setStakingContract()`
- `toggleStakingBonusRequirement()`
- `claimVestedTokens()`

However, these key state changes do not emit any events. The absence of an event makes it difficult to track admin-specific actions off-chain, which reduces transparency and complicates monitoring and auditing. It is recommended to emit events for this operation to ensure proper visibility, support off-chain indexing, and facilitate debugging, especially for critical administrative and user-facing fund transfers.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider emitting events within the mentioned affected functions to enhance transparency and ensure reliable off-chain tracking of key state changes. Emitting events not only improves monitoring of critical storage variable updates but also supports indexing services, facilitates auditing, and provides better visibility into administrative actions.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), events are now correctly emitted within the previously affected functions.

F-2025-13966 - Lack of Two Step Ownership Pattern - Info

Description:

The `transferFoundationRole` function immediately revokes the role from the caller without verifying that the new address is capable of receiving it (e.g., not a contract without the ability to call functions back). Additionally, if the transfer fails or the new address is compromised, there's no way to recover.

AlberichToken.sol:

```
function transferFoundationRole(address newDAO) external onlyFoundation {
    require(newDAO != address(0), "zero DAO");
    _grantRole(FOUNDATION_ROLE, newDAO);
    _revokeRole(FOUNDATION_ROLE, msg.sender);
    emit FoundationRoleTransferred(msg.sender, newDAO);
}
```

Impact:

- If `newDAO` is a contract without proper role management, foundation control is permanently lost
- No way to recover if `newDAO` is compromised immediately after transfer
- Single-step transfer is risky for critical admin roles
- No check if `newDAO` is already the foundation (unnecessary state change)

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Mitigated

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Implement two-step role transfer pattern:

```
address public pendingFoundation;

function initiateFoundationTransfer(address newDAO) external onlyFoundation {
    require(newDAO != address(0), "zero DAO");
    require(newDAO != msg.sender, "already foundation");
    pendingFoundation = newDAO;
    emit FoundationTransferInitiated(msg.sender, newDAO);
}

function acceptFoundationRole() external {
    require(msg.sender == pendingFoundation, "not pending");
    address oldFoundation = getRoleMember(FOUNDATION_ROLE, 0); // Get current
    _grantRole(FOUNDATION_ROLE, msg.sender);
    _revokeRole(FOUNDATION_ROLE, oldFoundation);
    pendingFoundation = address(0);
    emit FoundationRoleTransferred(oldFoundation, msg.sender);
}
```

Resolution: The two-step ownership transfer mechanism was implemented via the `initiateFoundationTransfer()` and `acceptFoundationRole()` functions in the second version of the provided file (SHA3-256: `a26e0a4`),. However, it is still possible for an account holding `DEFAULT_ADMIN_ROLE` to grant or revoke the `FOUNDATION_ROLE` directly via the inherited `grantRole()` and `revokeRole()` functions from the `AccessControl` library.

This creates a severe inconsistency in the foundation role management flow and may introduce unexpected or contradictory state transitions.

F-2025-13967 - Unbounded Loop in setWhitelist Allows Gas Griefing - Info

Description:

The `setWhitelist` function accepts an unbounded array and loops through it without gas limit checks. A foundation admin could accidentally (or maliciously) pass a very large array, causing the transaction to run out of gas and potentially making the function unusable.

AlberichToken.sol:

```
function setWhitelist(address[] calldata addrs, bool allowed) external onlyFo
undation {
    require(!parametersFrozen, "params frozen");
    for (uint256 i; i < addrs.length; ++i) {
        whitelist[addrs[i]] = allowed;
    }
}
```

As a result, large array can cause out-of-gas errors.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Consider adding array length limit to the function to handle large array operations in different batches.

Resolution: The issue was fixed in the second version of the provided file (SHA3-256: `a26e0a4`), array length limit was added to the `setWhitelist()` function:

```
uint256 public constant MAX_WHITELIST_BATCH = 500;

function setWhitelist(address[] calldata addrs, bool allowed) external onlyFo
undation {
    require(!parametersFrozen, "params frozen");
    require(addrs.length <= MAX_WHITELIST_BATCH, "batch too large");
    for (uint256 i; i < addrs.length; ++i) {
        whitelist[addrs[i]] = allowed;
    }
    emit WhitelistUpdated(addrs.length, allowed);
}
```

F-2025-13969 - Staking Placeholder Functions Are Incomplete and Confusing - Info

Description: The contract includes staking-related variables and functions (`stakingContract`, `stakingBonusRequired`, `isEligibleForBonus`) that are incomplete and not integrated with any actual functionality. The `isEligibleForBonus` function has confusing logic.

AlberichToken.sol:

```
function setStakingContract(address staking) external onlyFoundation { stakin
gContract = staking; }
function toggleStakingBonusRequirement(bool required_) external onlyFoundatio
n { stakingBonusRequired = required_; }
function isEligibleForBonus(address /*account*/) external view returns (bool)
{
    if (!stakingBonusRequired) return true;
    return (stakingContract != address(0)); // This doesn't check account's s
taking status!
}
```

Impact:

- `isEligibleForBonus` ignores the `account` parameter entirely
- Function returns **true** if staking contract is set, regardless of whether the account is actually staking
- Dead code increases contract size and deployment costs
- Confusing for users/integrators who might rely on this function

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: Either remove the placeholder functionality entirely, or implement it properly according to the desired logic.

Resolution: The issue was accepted. The functionality of the `isEligibleForBonus()` function still operates as a global staking eligibility check. The appropriate comment was added:

```
/**
* @notice Current placeholder: bonus eligibility is global, gated by whether staking is required
* and a stakingContract is configured. Per-account staking checks may be integrated later.
*/
```

However, the comment notes that per-account staking checks may be added in the future, while the contract itself is not upgradeable, meaning no such implementation changes can be introduced after deployment.

F-2026-17751 - Lack of Dedicated Event for Foundation Allocation Transfers - Info

Description:

The `transferAllocation()` function allows the foundation to move unlocked tokens from the contract's balance to a specified address. However, the function does not emit a dedicated event to distinguish these privileged allocation transfers from ordinary ERC-20 transfers. The only event emitted is the standard `Transfer` event triggered internally by `_transfer()`.

```
function transferAllocation(address to, uint256 amount)
    external
    onlyFoundation
{
    require(to != address(0), "zero addr");
    require(amount > 0, "zero amount");

    uint256 unlockedBalance = balanceOf(address(this)) - reservedForVesting;
    require(amount <= unlockedBalance, "insufficient unlocked");

    _transfer(address(this), to, amount);
}
```

This makes foundation-initiated allocation transfers indistinguishable from other token movements originating from the contract address (e.g., vesting claims, token purchases) when monitoring on-chain events. Off-chain indexers, dashboards, and governance monitoring tools cannot programmatically differentiate between these distinct operational flows.

This leads to following consequences:

- Reduced transparency and auditability of privileged foundation operations.
- Off-chain monitoring systems and block explorers cannot distinguish allocation transfers from vesting distributions or other contract-originated transfers.
- Complicates post-incident forensic analysis and real-time governance oversight.

Assets:

- src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]

Status: Fixed

Classification

Impact Rate:	1/5
Likelihood Rate:	4/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation: Define and emit a dedicated event that captures the full context of the allocation transfer:

```
event AllocationTransferred(address indexed to, uint256 amount);
```

Emit it within `transferAllocation()` after the transfer succeeds:

```
_transfer(address(this), to, amount);
emit AllocationTransferred(to, amount);
```

This enables off-chain systems to subscribe specifically to foundation allocation movements, improving operational transparency and monitoring fidelity.

Resolution: **Fixed in `d935eec`.**

The `transferAllocation()` function in **AlberichToken** now emits a dedicated `AllocationTransferred(address indexed to, uint256 amount)` event after executing the token transfer. The event declaration was added to the contract's event block, providing on-chain traceability for foundation allocation movements distinct from standard ERC-20 `Transfer` events.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
File Name	Alberich-Token_Internal_Test_Results_Final_2025-10-29.zip
File Hash (SHA256)	98c4c5fb3d0477249ca8ba4fcffa275c17a9f565d175ec5b76e8c2127d792c3f
Final File Name	UPDATED_Alberich_Token_Audit_Ready_Smart_Contract_With_Fixes_2025-11-14.sol
Final Fila Hash (SHA256)	a26e0a4d05b91f5fa2bee218f024c61df4f3b6189df1976157898cf674b75bb4
Additional Review File Name	UPDATED_Alberich_Token_Audit_Ready_Contract_With_Allocation_Fix_2026-05-27.sol
Additional Review File Hash (SHA256)	d395b4d0f052e1ae299021cd44ae7f4a5d12ee6d7c43e635116c53bb46ea921a
Additional Review Final File Name	UPDATED_Alberich_Token_Audit_Ready_Contract_Remediation_2026-06-04.sol
Additional Review Final File Hash (SHA256)	d935eec51449f8890722adcba7f2438b91ee1b2fceed2459c437acdc7b9427d7
Whitepaper	5. Smart-Contract_to_Whitepaper_Alignment_Report_v3.0.pdf
Requirements	2. AlberichToken_README_Audit_Instructions_FINAL_2025-10-29 (1)
Technical Requirements	4. Overview_Alberich_Token (ALBRH)_2025-10-29

Asset	Type
src/2. AlberichToken_AuditReady_SmartContract_FINAL (10-13-2025).sol [-]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of Alberich Token, Hacken followed its methodology by performing fuzz-testing on the project's main functions. Foundry, a tool used as a testing framework, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Invariant	Test Result
testFuzz_PurchaseCalculation_NoOverflow	Passed
testFuzz_PurchaseCalculation_Correctness	Passed
testFuzz_PurchaseFloorDivision	Passed
testFuzz_SHOCap_NeverExceeded	Passed
testFuzz_PublicCap_NeverExceeded	Passed
testFuzz_MinPurchase_Enforced	Passed
testFuzz_Vesting_LinearRelease	Passed
testFuzz_Vesting_Monotonic	Passed
testFuzz_Vesting_ReleasedBounded	Passed
testFuzz_ExtremePrice_VeryLow	Passed
testFuzz_ExtremePrice_VeryHigh	Passed
testFuzz_DustAccumulation	Passed
testFuzz_PurchaseAtCapBoundary	Passed
testFuzz_VestingBoundaries	Passed

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.