ALMANAK

SMART CONTRACTS REVIEW

zokyo

May 6th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING ALMANAK

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 1 Critical issue: 1 resolved = 0 points deducted
- 1 High issue: 1 resolved = 0 points deducted
- 1 Medium issue: 1 resolved = 0 points deducted
- 3 Low issues: 3 resolved = 0 points deducted
- 2 Informational issues: 2 acknowledged = 0 points deducted

Thus, the score is 100

TECHNICAL SUMMARY

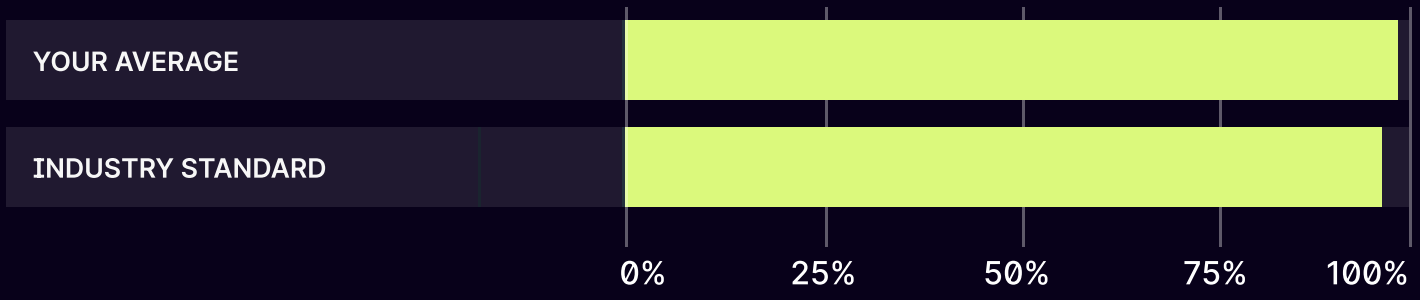
This document outlines the overall security of the Almanak smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Almanak smart contract/s codebase for quality, security, and correctness.

Contract Status

There was 1 critical issues found during the review. (See Complete Analysis)

Testable Code



100% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Almanak team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Almanak repository:
Repo: https://github.com/almanak-co/contracts/commit/0068eab1cd4f26a16d4f96284d07006aa6abf032

Last commit - 321b345

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./LockedAlmanakToken.sol
- ./AlmanakVestingWallet.sol
- ./VestingWalletFactory.sol
- ./AlmanakToken.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Almanak smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Fuzz tested the contract to detect logic issues via randomized inputs.
04	Thorough manual review of the codebase line by line.

Executive Summary

The **Almanak audit** comprised four smart contracts: `AlmanakToken.sol`, `AlmanakVestingWallet.sol`, `LockedAlmanakToken.sol`, and `VestingWalletFactory.sol`.

- The **AlmanakToken** is a simple smart contract that functions as a standard ERC20 token.
- The **LockedAlmanakToken** is a soulbound governance token intended for off-chain voting, minted 1:1 with locked tokens.
- Non-transferable by users.
- Multiple vesting contracts can mint or burn tokens if authorized by the owner.

The **VestingWalletFactory** is a minimal proxy factory that deploys clones of the `AlmanakVestingWallet` contract, all sharing a single instance of `LockedAlmanakToken`. This factory contract also allows the factory owner to trigger token removal on a specific vesting wallet through the `removeTokens` function.

The **AlmanakVestingWallet** is a vesting contract that mints and burns tokens from a shared `LockedAlmanakToken` instance.

- The soulbound token contract must designate this contract as an authorized minter. The underlying (locked) token is `AlmanakToken`, or any standard ERC20 token.

Anyone can deposit locked tokens into this contract, while the vesting wallet owner can claim the released tokens according to the vesting schedule.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Almanak team and the Almanak team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Users can claim their whole vested amount without waiting the whole duration period	Critical	Resolved
2	Release Function Lacks Access Control Leads To Anyone Releasing Tokens At Arbitrary Time Intervals Which Can Be Incorrect For Wallet Owners	High	Resolved
3	Users can add released tokens after cliff to their own vesting schedule to increase the release rate	Medium	Resolved
4	The `removeTokens` function can be executed on any address	Low	Resolved
5	Ownership of the contract can be transferred to address(0)	Low	Resolved
6	Make The Transfer Of Ownership A 2 Step Process	Low	Resolved
7	Allowed actors can burn tokens from other users	Informational	Acknowledged
8	The owner keys in the system holds significant privileges	Informational	Acknowledged

Users can claim their whole vested amount without waiting the whole duration period

Description:
The `releasable()` function within the `AlmanakVestingWallet.sol` smart contract calculates the amount of vested tokens that can be released at the current time:

```
function releasable(address token) public view virtual override returns
(uint256) {
        require(token == address(lockedToken), "Unsupported token");

        // Amount vested according to schedule (based on _totalDeposited)
        uint256 vested = _vestedAmount(token, uint64(block.timestamp));
        // Amount actually available in the contract
        uint256 balance = lockedToken.balanceOf(address(this));

        // Return the minimum of what's vested and what's available
        return vested < balance ? vested : balance;
    }
```

However, this function is not accounting the already released amounts.

The release of locked tokens inside the wallet contract is supposed to be done linearly over time i.e. when release() is invoked in the AlmanakVestingWallet.sol it calculates how much tokens to release based on how much time has elapsed since the start , but it fails to account for the tokens already released resulting in non linear and incorrect releases.
For example - If the duration of the vest was 4 years and 1000 tokens were deposited then after first year the released tokens would be →

```
(basis * (timestamp - start())) / duration()
```

= 1000 * 1 year / 4 year = 250

For the second year →

= 1000 * 2 year / 4 year = 500 tokens , which is incorrect since 250 should have been released according to the linear flow.

The core issue is that releasable() has been overriden from the OZ library but the deduction of released() is missing in the almanak implementation.

Impact:
Users can invoke the releasable() function repeatedly without restriction. However, the function does not account for the amount that has already been released. As a result, if a user makes their first claim halfway through the vesting duration, they will receive 50% of the total allocation. If they immediately claim again, they will receive the remaining 50%, effectively bypassing the intended vesting schedule and obtaining the full amount prematurely.

Proof of concept POC: https://gist.github.com/JMariadlcs/b32cecae42957f2f3f09e11ad46ae120

Recommendation:
It should be tracked the already released amount by adding a `__released` variable within the release function which grows every time an amount is released:

```
__released += actuallyReleased;
```

It should be also discounted this amount from the releasable amount calculated within `releasable()`:

```
uint256 vested = _vestedAmount(token, uint64(block.timestamp)) -
  released;
```

HIGH-1 | RESOLVED

Release Function Lacks Access Control Leads To Anyone Releasing Tokens At Arbitrary Time Intervals Which Can Be Incorrect For Wallet Owners

Almanak creates different wallet contracts according to different business logic needs of users , for example → there can be a requirement that a user wants to have a wallet where the stake should be released after every 3 months , but since the release() function in the wallet is public anyone can release the staked tokens which would lead to irregular token releases.

Recommendation:
Only let the owner release the vested tokens.

sers can add released tokens after cliff to their own vesting schedule to increase the release rate.

Description:
The `addTokens()` function within the `AlmanakVestingWallet.sol` smart contract is used to add the locked tokens to the contract which are going to be distributed within the vesting schedule. This function can be called by anyone at any point in time, even after the cliff:

```
function addTokens(uint256 amount) external {
        require(
            lockedToken.transferFrom(msg.sender, address(this), amount),
            "Transfer failed"
        );
        // Update the high-water mark
        _totalDeposited += amount;

        // Now that 'amount' more tokens are locked, mint soulbound
tokens
        // This requires that this contract has 'isMinter[this] = true'
in sbToken
        sbToken.mint(owner(), amount); // @audit is it correct to mint to
owner()
    }
```

Impact
Since the function is declared as `public`, it can be executed by any user. A recipient of the released tokens could potentially invoke the `addTokens()` function to redeposit those tokens, thereby artificially inflating the `_totalDeposited` value.
The `_totalDeposited` is used to calculate the releasable amount at any moment:

```
uint256 basis = _totalDeposited;
return (basis * (timestamp - start())) / duration();
```

This means that users can inflate this variable in order to increase their releasable speed.

Recommendation:
If the primary purpose of this function is to allow only the vesting contract creator to deposit the tokens intended for distribution, it is recommended to restrict access to this function using an appropriate access control modifier. This would prevent unauthorized actors from re-depositing tokens and creating inconsistencies in the `_totalDeposited` accounting.

The `removeTokens` function can be executed on any address

Description:
The `executeRemoveTokens()` function within the `VestingWalletFactory.sol` smart contract allows the factory owner to trigger token removal on a specific vesting wallet by executing the following call:

```
function executeRemoveTokens(address payable wallet, uint256 amount)
external onlyOwner {
        // TODO: Consider adding validation to ensure 'wallet' is
actually one created by this factory.
        // Call removeTokens on the target wallet. The Factory contract
is msg.sender here.
        AlmanakVestingWallet(wallet).removeTokens(amount);
    }
```

However, it is not checked wether `wallet` is one of the previously created clones by the contract.

Impact:
If the `wallet` is set to an address that does **not** belong to a previously deployed clone, and this address points to a contract that exposes a public function—or any function callable by the factory—that ultimately triggers `removeTokens()`, then arbitrary behavior defined in that contract can be executed.

This issue is marked as **low severity** because the `executeRemoveTokens()` function is protected by an `onlyOwner` modifier. However, it still presents a potential risk if the owner mistakenly sets an incorrect address or if the owner's account is compromised.

Recommendation:
Include a ‘check’ to ensure that ‘wallet’ is part of the created clones by the same smart contract.

LOW-2 | RESOLVED

Ownership of the contract can be transferred to address(0)

Description:
The `LockedAlmanakToken.sol` smart contract implements a `transferOwnership()` function which is only executable by the owner. The goal of this function is to update the current owner of the smart contract. However, it is not checked if the new owner is set to address(0).

```
function transferOwnership(address newOwner) external onlyOwner {
        owner = newOwner;
    }
```

Impact:
Setting the owner to address(0) would be the same as 'loosing' the ownership. This means that ' onlyOwner' functions are not longer going to be available. Therefore, changing again the owner is neither going to be possible.

Recommendation:
Implement a check to ensure that the new owner is not `address(0)`.

```
require(newOwner != address(0), "Invalid owner");
```

LOW-3 | RESOLVED

Make The Transfer Of Ownership A 2 Step Process

The transferOwnership function is used to change the existing owner in the LockedAlmanakToken.sol , ideally this should be a 2 step process where a pendingOwner should be set first and a confirmOwnership function where the pendingOwner accepts the owner role , this is because in the current implementation if the owner address assigned is incorrect then ownership is lost forever.

Recommendation:
Make the transfer of ownership a 2 step process.

INFORMATIONAL-1 | ACKNOWLEDGED

Allowed actors can burn tokens from other users

Description:
Certain allowed actors, which are set as `isMinter` by the owner within the `LockedAlmanakToken.sol` smart contract are able to burn tokens from other users. This is just an informational issue to inform users about this type of behavior, which may be expected.

```
/**
     * @notice Burn soulbound tokens from `account`.
     * @dev Only authorized minters (vesting contracts) can call this.
     */
    function burn(address account, uint256 amount) external onlyMinter {
        _burn(account, amount);
    }
```

INFORMATIONAL-2 | ACKNOWLEDGED

The owner keys in the system holds significant privileges

Description:
A specific key in the system holds significant privileges, including the ability to deploy schedules and potentially enable or execute revocations. Given the level of control associated with this key, improper handling or storage poses a potential risk to the overall system security.

Impact:
While no direct vulnerability is currently exploitable, the misuse or compromise of this high-privilege key could lead to critical consequences, including unauthorized schedule deployments or unintended revocations. This makes it a target for attackers and a potential point of failure in case of mismanagement.

Recommendation:
Secure this key following best practices for high-privilege access, such as using a multisignature wallet, hardware wallet, or other secure key management mechanisms. Additionally, access to the key should be regularly reviewed and limited to only what is strictly necessary.

	./LockedAlmanakToken.sol ./AlmanakVestingWallet.sol ./VestingWalletFactory.sol ./AlmanakToken.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Fuzz tests written by Zokyo Security

As part of our work assisting Almanak in validating their smart contract/s, we performed fuzz testing using the Foundry framework. The goal was to identify potential edge cases or unexpected behaviors by automatically generating a wide range of randomized inputs against the contract's functions.

Ran 8 tests for .audit/test/LockedAlamanakTokenFuzzTest.t.sol:LockedAlmanakTokenFuzzTest
[PASS] testFuzz_Burning(address,address,uint256) (runs: 1024, μ: 94403, ~: 95694)
[PASS] testFuzz_Minting(address,address,uint256) (runs: 1024, μ: 49097, ~: 46811)
[PASS] testFuzz_TransferShouldAlwaysFail(address,address,uint256) (runs: 1024, μ: 93099, ~: 93099)
[PASS] testFuzz_addMinterCaseNonOwner(address) (runs: 1024, μ: 14771, ~: 14771)
[PASS] testFuzz_removeMinter(address,address,address) (runs: 1024, μ: 22404, ~: 20757)
[PASS] testFuzz_transferOwnership(address,address) (runs: 1024, μ: 18840, ~: 18837)
[PASS] test_Minting() (gas: 86404)
[PASS] test_MintingRevert() (gas: 45277)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 910.68ms (3.20s CPU time)

Ran 7 tests for .audit/test/VestingWalletFactoryFuzzTest.t.sol:VestingWalletFactoryTest
[PASS] testFuzz_CreateMultipleWalletsWithVaryingParams(uint64[3],uint64[3],bool[3]) (runs: 1024, μ: 809466, ~: 810534)
[PASS] testFuzz_CreateVestingWallet(uint64,uint64,uint64,bool) (runs: 1024, μ: 306876, ~: 306684)
[PASS] testFuzz_ExecuteRemoveTokensVaryingAmounts(uint256,uint256) (runs: 1024, μ: 464923, ~: 466302)
[PASS] test_RevertWhenTokensRemovable() (gas: 424924)
[PASS] test_VerifySetup() (gas: 23432)
[PASS] test_WithdrawTokens() (gas: 465002)
[PASS] test_WithdrawTokens(address) (runs: 1024, μ: 343112, ~: 344330)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 5.47s (11.84s CPU time)

Ran 15 tests for .audit/test/AlmanakVestingWalletFuzzTest.t.sol:AlmanakVestingWalletFuzzTest
[PASS] testFuzz_AddTokens(uint256) (runs: 1024, μ: 167653, ~: 167053)
[PASS] testFuzz_BadReleasable(address) (runs: 1024, μ: 18983, ~: 18983)
[PASS] testFuzz_CliffPeriod(uint64) (runs: 1024, μ: 182586, ~: 181970)
[PASS] testFuzz_CombinedOperations(uint64,uint256,uint256,uint64,uint256) (runs: 1024, μ: 301059, ~: 308057)

```
[PASS] testFuzz_MultipleTokenAdditions(uint256[]) (runs: 1024, μ: 208746, ~: 209523)
[PASS] testFuzz_Release(uint64) (runs: 1024, μ: 224665, ~: 224756)
[PASS] testFuzz_RemoveTokensCaseA(uint256) (runs: 1024, μ: 200618, ~: 200233)
[PASS] testFuzz_RemoveTokensCaseB(uint256) (runs: 1024, μ: 183705, ~: 196738)
[PASS] testFuzz_RemoveTokensCaseC(uint256) (runs: 1024, μ: 174424, ~: 173772)
[PASS] testFuzz_RevertAddTokensWithBadTransfer(uint256) (runs: 1024, μ: 93620, ~: 92968)
[PASS] testFuzz_VestingScheduleOverTime(uint64) (runs: 1024, μ: 162128, ~: 161150)
[PASS] test_RevertWhen_NonFactoryRemovesTokens() (gas: 147502)
[PASS] test_RevertWhen_ReleasingUnsupportedToken() (gas: 858863)
[PASS] test_RevertWhen_RemovingTokensWhenNotRemovable() (gas: 2470345)
[PASS] test_VerifySetup() (gas: 56386)
```

We are grateful for the opportunity to work with the Almanak team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Almanak team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo