QuillAudits

AUDIT REPORT

December 2025

For

ALMANAK

Table of Content

Executive Summary

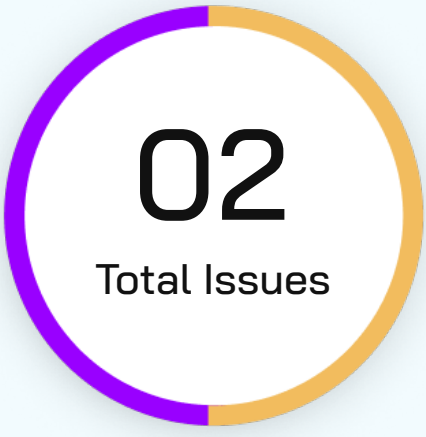
Project Name	Almanak
Protocol Type	Airdrop, Escrow
Project URL	https://almanak.co
Overview	A Merkle-based airdrop contract with optional slashing that decreases over time. Supports claiming or claim-and-stake into veToken, with emergency pause and post-deadline revocation. Airdrop where staking commitment (percentage + unlock time) is pre-encoded in the Merkle tree. Staked portion is slash-free, while unstaked portion faces time-decaying slashing. Vote-escrow (ve) system where ALMANAK tokens are locked for up to 2 years to gain decaying voting power. Supports authorized contracts to create locks for users (airdrop integration) with full supply and history checkpoints. Distributes fees weekly to veToken holders based on historical voting power snapshots. Users claim proportional rewards, with gas-safe checkpointing and epoch-based tracking.
Audit Scope	The scope of this Audit was to analyze the Almanak Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/almanak-co/contracts
Contracts In Scope	AlmanakAirdrop.sol VotingEscrowALMANAK.sol AlmanakAirdropWithPreStake.sol FeeDistributor.sol
Branch	Main
Commit Hash	3c4b5e060550483f97d2e8ad935043975fc46b49
Language	Solidity
Blockchain	Ethereum

Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	24th November - 8th December 2025
Updated Code Recieved	9th December 2025
Review 2	9th December 2025
Fixed In	0eb29569ca1f718750b7e135a2ef1f651d40f0e7

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0(0.0%)
High	0(0.0%)
Medium	0(0.0%)
Low	1 (50.0%)
Informational	1 (50.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	**1**	**1**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Use Ownable2Step Instead of Ownable	Low	Resolved
2	Incorrect Accounting When Using Fee-on-Transfer Tokens	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Low Severity Issues

Use Ownable2Step Instead of Ownable

Resolved

Path

AlmanakAirdropWithPreStake.sol
AlmanakAirdrop.sol
VotingEscrowALMANAK.sol

Description

The project intends to use Ownable2Step semantics (safe 2-step ownership transfer), but currently inherits from Ownable, not Ownable2Step.
This means the owner can instantly transfer ownership in one transaction defeating the intended Two-Step security model documented in comments and project requirements.

If the owner key is compromised or misused, privileged control can be reassigned instantly without the beneficiary's acceptance.

Impact

Low

Likelihood

Low

Recommendation

Replace Ownable with Ownable2Step to enforce a safer two-step ownership transfer process.

Informational Issues

Incorrect Accounting When Using Fee-on-Transfer Tokens

Resolved

Path

FeeDistributor.sol

Function Name

`_claim()`

Description

The contracts assume the token being distributed is a standard ERC-20 where the transferred amount equals the amount credited.
With Fee-on-Transfer (FoT) tokens, the contract receives or sends less tokens than expected due to burn/tax mechanics.

This causes:

- Incorrect totalClaimed, totalStaked, and reward accounting
- veToken supply inflated (deposit value > real transferred value)
- Reward distribution mismatch in FeeDistributor

State variables become desynced from actual balances, resulting in early depletion or inability to distribute tokens fairly.

Impact

Medium

Likelihood

Low

Recommendation

Use balance-diff accounting

Functional Tests

AlmanakAirdrop

- Claims execute only during the active window with a valid proof, transferring the correct user amount while routing any slashed portion to the slashing receiver.
- claimAndStake correctly locks the committed portion without slashing, enforces unlock-time requirements, and sends the remaining liquid tokens to the recipient.
- Pausing stops all claim operations and only the owner can pause or resume the contract.
- After the deadline, unclaimed tokens can be revoked a single time by the owner and safely transferred to the chosen recipient.

AlmanakAirdropWithPreStake

- Claim-and-stake works only inside the claim window and requires a valid Merkle commitment containing amount, stake percentage, and unlock time.
- Stake percentage rules are properly enforced and only the unstaked portion undergoes time-based slashing.
- totalStaked, totalClaimed and hasClaimed accurately reflect one-time usage per eligible address.
- Unlock time must outlast the slashing period and funds are split correctly between staked and liquid outputs.
- Owner-controlled operations (pause, unpause, revoke, update slashing receiver, recover tokens) behave correctly with proper restrictions.

FeeDistributor

- Reward checkpointing properly allocates newly deposited tokens across weeks based on elapsed distribution time.
- Weekly supply snapshots capture accurate voting power values from the veToken contract.
- Reward claiming distributes proportional weekly payouts up to the user's claim cursor or iteration limit.
- No payout occurs when weekly rewards are zero or user voting power is zero.

VotingEscrowALMANAK

- Lock creation accepts only a non-zero amount, a future week-aligned unlock time, and no existing active lock for that address.
- Voting power follows correct linear decay over time and adjusts immediately on deposits or duration extensions.
- Lock modifications respect expiry and maximum lock duration rules without corrupting state.
- Withdrawals release the full locked token amount only once the lock has expired.
- createLockFor operates exclusively for authorized creators, and authorization can only be modified by the owner.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats

VotingEscrowALMANAK.sol	createLock(uint256 _value, uint256 _unlockTime)	Allows user to create a new lock and receive voting power.
		
		Inputs
		
		_value
		
		• Control: Fully controlled by the caller (msg.sender).
		• Constraints:
		  • Must be > 0 (AmountTooSmall).
		  • Caller must have at least _value tokens and have approved the VE contract to pull them (via safeTransferFrom in _depositFor).
		• Impact:
		  • Amount of tokens being locked.
		  • Directly affects user's slope/bias and ve-balance.
		  • Affects global supply and total voting power.
		
		_unlockTime
		
		• Control: Fully controlled by the caller.
		• Constraints:
		  • Internally rounded down:
		  • unlockTime = (_unlockTime / WEEK) * WEEK.
		  unlockTime > block.timestamp (UnlockTimeNotInFuture).
		  unlockTime <= block.timestamp + MAXTIME (LockTimeTooLong).
		  Caller must not already have an active lock (LockAlreadyExists).

Contract	Function	Threats

		• Impact:
		  • Determines lock end time, slope, and initial bias.
		  • Misuse (passing block.timestamp or short future times) leads to revert due to rounding down to a past week.
		
		Branches and code coverage
		
		Intended branches
		1. First lock creation
		  • _value > 0
		  • No existing lock (locked[msg.sender].amount == 0).
		  • unlockTime (after rounding) is strictly in the future and within MAXTIME.
		  • _depositFor mints voting power and updates supply and checkpoints correctly.
		2. Global state updates
		  • supply increases by exactly the deposited amount.
		  • Global and user Point histories updated via _checkpoint.
		
		Test coverage
		• Should create a new lock and set:
		  • locked[msg.sender].amount == _value
		  • locked[msg.sender].end == roundedUnlockTime
		• Should mint ve-balance (non-zero balanceOf(msg.sender)).
		• Should update total supply (totalSupply() > 0).

Contract	Function	Threats

		• Should handle rounding to week correctly:
		  • createLock(amount, block.timestamp + N*WEEK) succeeds.
		  • createLock(amount, block.timestamp + 3 days) reverts with UnlockTimeNotInFuture (given current floor behavior).
		
		Negative behavior
		• Should not allow zero amount (_value == 0 → AmountTooSmall).
		• Should not allow creating a new lock if one exists (LockAlreadyExists).
		• Should not allow past or “non-future after rounding” unlock times (UnlockTimeNotInFuture).
		• Should not allow lock longer than MAXTIME (LockTimeTooLong).
		• Should be safe under fee-on-transfer tokens (if you choose to support them) or explicitly revert if FOT is used.

Contract	Function	Threats

		Intended branches
		• createLock(0, futureTime) → revert AmountTooSmall.
		• Two consecutive createLock calls without withdraw → revert
		• LockAlreadyExists. createLock(amount, block.timestamp) → revert UnlockTimeNotInFuture.
		• createLock(amount, block.timestamp + MAXTIME + 1) → revert LockTimeTooLong.
		• For FOT token mock:
		  • If VE is FOT-safe: ensure supply and locked.amount match received, not _value.
		  • If not supporting FOT: enforce revert and test for it.

	increaseAmount(uint256 _value)	**Allows user to add more tokens to an existing lock.**
		Inputs
		_value
		• Control: Fully controlled by the caller
		• Constraints:
		  • _value > 0 (AmountTooSmall).
		  • Existing lock must exist (NoLockFound).
		  • Lock must be active: locked.end > block.timestamp (CannotAddToExpiredLock).
		• Impact:
		  • Increases locked[msg.sender].amount. Increases slope and bias (more voting power).
		  • Increases global supply

Contract	Function	Threats
		Branches and code coverage Intended branches 1. Add to active lock • locked.amount > 0, locked.end > now, _value > 0. • _depositFor updates amount and recalculates slope/bias. 2. Time-dependent decay • Check that calling increaseAmount after some time results in correct remaining voting power + increment. Test coverage • Should revert if no lock exists. • Should revert if lock is expired. • Should increase locked.amount by _value. • Should increase supply and user's ve-balance appropriately. **Negative behavior** • Zero _value → AmountTooSmall. • Expired lock → cannot add (CannotAddToExpiredLock). • No pre-existing lock → NoLockFound. • FOT token mismatch same as above. Negative tests • increaseAmount(0) with active lock → AmountTooSmall. • increaseAmount(amount) with no lock → NoLockFound. • increaseAmount(amount) after vm.warp(locked.end + 1) → • CannotAddToExpiredLock.

Contract	Function	Threats

	increaseUnlockTime(uint256 _unlockTime)	Allows user to extend the lock's end time.
		
		Inputs
		
		_unlockTime
		
		• Control: Fully controlled by caller.
		• Constraints:
		  • Rounded: unlockTime = (_unlockTime / WEEK) * WEEK.
		  • Must have existing lock (NoLockFound).
		  • Lock must be active (CannotAddToExpiredLock).
		  • New unlock must be strictly greater than current locked.end (CanOnlyIncreaseLockDuration).
		  • Must not exceed now + MAXTIME (LockTimeTooLong).
		• Impact:
		  • Extends time-remaining -> increases bias / voting power duration.
		  • Updates slope & slopeChanges schedule.
		
		Branches and code coverage
		
		Intended branches
		
		1. Extend unlock time forward (happy path)
		  • unlockTime rounded forward (after user's input) and strictly > old end.
		2. Calling with same or lower unlockTime
		  • Should revert with CanOnlyIncreaseLockDuration.

Contract	Function	Threats
		Test coverage • Should extend lock end and keep amount unchanged. • Should recompute slope/bias. • Should update slopeChanges at old and new lock end times. **Negative behavior** • No lock / expired lock -> cannot extend. • unlockTime <= current end -> revert. • unlockTime > now + MAXTIME -> revert. **Negative tests** • increaseUnlockTime(newTime <= currentEnd) → • CanOnlyIncreaseLockDuration. increaseUnlockTime when lock expired → CannotAddToExpiredLock.

	withdraw()	**Allows user to withdraw tokens after lock expiry.**
		Inputs
		• No explicit params; uses locked(msg.sender).
		Behavior
		• Constraints:
		  • Must have a lock (NoLockFound).
		  • block.timestamp >= locked.end (LockNotExpired).

Contract	Function	Threats

		• Impact:
		  • Transfers underlying tokens back to user.
		  • Sets locked.amount = 0, locked.end = 0.
		  • Reduces global supply.
		  Updates user and global Point history.
		
		Branches and code coverage
		
		Intended branches
		1. Withdraw after expiry (happy path)
		  • locked.amount > 0, now ⩾ end.
		2. Withdraw before expiry
		  • Revert with LockNotExpired.
		3. Supply update
		  • supply decreases by value. _checkpoint called to decay and remove user weight.
		
		Test coverage
		• Should revert if no lock.
		• Should revert if now < end.
		• Should return full locked token amount.
		• Should zero out locked storage.
		• Should reduce supply and ve balance to zero.
		
		Negative behavior
		• Attempt early withdrawal — must revert.
		• Ensure no rounding / underflow in supply = supplyBefore - value.
		• Ensure withdrawal doesn't break if user had 0 slope (e.g. after _checkpoint drift).

Contract	Function	Threats
		Negative tests • Withdraw before locked.end → LockNotExpired. • Withdraw multiple times (second time should revert NoLockFound).

	createLock(uint256 _value, uint256 _unlockTime)	**Allows authorized creator (e.g., airdrop contract) to create a lock on behalf of user _for.**
		Inputs
		_for
		• Control: Fully controlled by authorized caller.
		• Constraints:
		  • Can be any address (non-zero strongly recommended).
		  • That address must not already have a lock (LockAlreadyExists).
		• Impact:
		  • _for becomes owner of the lock (ve power).
		  • Caller supplies the tokens (transferFrom msg.sender).
		_value
		• Control: Controlled by creator.
		• Constraints:
		  • _value > 0.
		  • Creator must hold _value tokens & approve VE contract.
		• Impact:
		  • Lock balance for _for.
		  • Increases global supply.

Contract	Function	Threats

		_unlockTime
		• Same rounding/constraints as createLock:
		  • unlockTime > now, unlockTime <= now + MAXTIME.
		
		Branches and code coverage
		Intended branches
		1. Authorized creator creates lock (happy path)
		  • isAuthorizedCreator(msg.sender) == true.
		  • _for has no lock.
		  • _value > 0, unlockTime in range.
		2. Unauthorized caller
		  • Revert with Unauthorized.
		
		Test coverage
		• Should succeed if msg.sender authorized and _for has no lock.
		• Should update locked(_for) and global supply.
		• Should transfer tokens from creator to VE contract.
		
		Negative behavior
		• Unauthorized caller → Unauthorized. Existing lock for _for → LockAlreadyExists.
		• _value == 0 → AmountTooSmall.
		• Past or too far future unlock time → reverts as in createLock.

Contract	Function	Threats
		Negative tests • Call from non-authorized address → Unauthorized. • Authorized creator tries to lock for _for who already has a lock → LockAlreadyExists
	checkpoint()	Records global checkpoint (decays bias/slope up to now, updates pointHistory and epoch). **Inputs** **None; uses global state.** **Behavior** • Calls _checkpoint(address(0), empty, empty) to advance global state. • No direct token movement, but affects readings of totalSupply() / totalSupplyAt(). **Branches and code coverage** Intended branches 1. Called periodically • Updates pointHistory until current block timestamp. 2. Called multiple times in same block • Should not break or double-count. Test coverage • Multiple checkpoint() calls in same block should be idempotent in terms of effective supply projection. • totalSupplyAt using epochs created by checkpoint() should match _totalSupply(now) for current time.

Contract	Function	Threats
		Negative behavior • Potential gas exhaustion if many weeks between checkpoints (loop up • to 255 weeks). But bounded by design. • Needs to be safe even if no active locks exist. **Negative tests** • Call checkpoint() with no locks → should not revert. • Call after long warp → ensure epoch increments and no underflow.

AlmankAirdrop.sol	claim(uint256 amount, address recipient, bytes32[] merkleProof)	**Allows a user to claim their airdrop allocation (subject to slashing), sending tokens directly to a recipient address.**
		Inputs
		amount
		• Control: Fully controlled by the caller.
		• Constraints:
		  • Must be > 0 (InvalidClaimAmount).
		  • Must match the value in the Merkle tree for msg.sender.
		• Impact:
		  • Determines how many tokens are claimed + how much is slashed.
		  • Directly increases totalClaimed and decreases “unclaimed” accounting.

Contract	Function	Threats

		recipient
		• Control: Fully controlled by the caller.
		• Constraints:
		  • Must be non-zero
		  • (InvalidTokenAddress).
		• Impact:
		  • Receives the net (post-slash) tokens.
		  • This can be the caller, an EOA, or a contract.
		
		merkleProof
		• Control: Provided by the caller; must be correct to claim.
		• Constraints:
		  • MerkleProof.verify(merkleProof, merkleRoot, leaf) must return true, where leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender, amount)))).
		• Impact:
		  • Proves user's entitlement; incorrect proof → revert InvalidProof.
		
		Branches and code coverage
		Intended branches
		1. Valid claim (happy path)
		  • Claim window open:
		  • block.timestamp >= claimStartTime
		  • block.timestamp <= claimDeadline
		  • Caller has not claimed before (hasClaimed[msg.sender] == false).
		  • amount > 0.
		  • recipient != address(0).

Contract	Function	Threats

		• Merkle proof valid. hasClaimed[msg.sender] set to true, totalClaimed += amount.
		• Slashing:
		  • slashingRate = calculateSlashingRate(now) (0–10000 bps).
		  • slashedAmount = amount * rate / 10000.
		  • userAmount = amount - slashedAmount.
		• Transfers:
		  • If slashedAmount > 0 → send to slashingReceiver.
		  • If userAmount > 0 → send to recipient.
		• Emit TokensClaimed.
		
		2. Edge: slashing disabled
		
		• If slashingPeriodDuration == 0 → calculateSlashingRate() always returns 0.
		• Then slashedAmount == 0, userAmount == amount.
		
		3. Edge: claim at/near boundaries
		
		• Exactly at claimStartTime → allowed.
		• Exactly at claimDeadline → allowed (> claimDeadline reverts).

Contract	Function	Threats

		Test coverage
		• Should allow valid claim during window with correct proof:
		  • hasClaimed(caller) becomes true.
		  • totalClaimed increases by amount.
		  • Net user tokens and slashed tokens match formula.
		• Should handle no slashing if:
		  • slashingPeriodDuration == 0, or
		  • timestamp >= claimStartTime + slashingPeriodDuration (rate is 0).
		• Should emit TokensClaimed with correct values:
		  • (claimant, recipient, amount, userAmount, slashedAmount).
		Negative behavior
		• Claim before claimStartTime → ClaimPeriodNotStarted.
		• Claim after claimDeadline → ClaimPeriodEnded.
		• Double claim by same address → AlreadyClaimed.
		• amount == 0 → InvalidClaimAmount. recipient == address(0) → InvalidTokenAddress.
		• Invalid merkle proof → InvalidProof.
		• Claims when contract is paused → revert via whenNotPaused.

Contract	Function	Threats
		Negative tests • Claim with correct proof but before claimStartTime → revert • ClaimPeriodNotStarted. Claim after claimDeadline → revert ClaimPeriodEnded. Claim twice (same address, same proof) → second call reverts AlreadyClaimed. Claim with amount = 0 → InvalidClaimAmount. Claim with recipient = 0 → InvalidTokenAddress. Claim with invalid proof → InvalidProof. Claim while pause()d → revert via whenNotPaused.

	claimAndStake(uint256 amount, uint256 stakeAmount, uint256 unlockTime, address recipient, bytes32[] merkleProof)	**Allows user to claim and optionally stake part of their allocation into veToken in a single transaction.**
		Inputs
		amount
		• Control: Fully controlled by caller.
		• Constraints:
		  • Must be > 0 (InvalidClaimAmount).
		  • Must match Merkle tree for msg.sender.
		• Impact:
		  • Total allocation for the caller.
		  • Splits into staked + unstaked amounts.

Contract	Function	Threats

		stakeAmount
		• Control: Fully controlled by caller.
		• Constraints:
		  • stakeAmount <= amount (StakeAmountExceedsClaimAmount).
		  • If stakeAmount > 0, unlockTime != 0 (UnlockTimeRequired).
		  • If stakeAmount > 0, unlockTime >= (claimStartTime + slashingPeriodDuration)
		  • (UnlockTimeBeforeSlashingPeriodEnds).
		• Impact:
		  • Portion of claim deposited into veToken.
		  • Increases ve lock but is not slashed.
		
		unlockTime
		• Control: User-controlled, but constrained.
		• Constraints:
		  • Only relevant if stakeAmount > 0.
		  • Must be non-zero (UnlockTimeRequired).
		  • Must be ⩾ slashingEndTime (UnlockTimeBeforeSlashingPeriodEnds).
		  • Additional constraints may come from veToken.createLockFor (e.g., rounding & max time).
		• Impact:
		  • Lock expiry in veToken for the staked portion.

Contract	Function	Threats

		recipient
		• Control: User-controlled.
		• Constraints:
		  • Must be non-zero (InvalidTokenAddress).
		• Impact:
		  • Receives the liquid (unstaked, post-slash) tokens.
		
		merkleProof
		• Same semantics as in claim.
		
		Branches and code coverage
		
		Intended branches
		1. Claim with no stake (stakeAmount == 0)
		  • stakeAmount == 0 → skip staking.
		  • unstaked = amount.
		  • slashedAmount = unstaked * rate / 10000.
		  • liquidAmount = unstaked - slashedAmount.
		  • Only liquid + slashed transfers.
		2. Claim with partial stake (0 < stakeAmount < amount)
		  • stakeAmount tokens → approved to veToken, then staked via createLockFor.
		  • unstaked = amount - stakeAmount.
		  • Slashing only on unstaked part.
		  • Liquid + slashed transfers as above.

Contract	Function	Threats

		3. Claim with full stake (stakeAmount == amount)
		• unstaked = 0 → no slashing or liquid transfer.
		• All tokens go to veToken lock for the user.
		4. Slashing dynamics
		• Early in slashing period → high rate (close to initialSlashingRate).
		• After slashing ends → slashingRate == 0, so unstaked portion is fully liquid.
		5. Interaction with veToken
		• token.forceApprove(veToken, stakeAmount) then
		• createLockFor(msg.sender, stakeAmount, unlockTime), then reset approval to 0.
		• Assumes:
		  • Airdrop contract has enough tokens.
		  • veToken treats this airdrop contract as an authorized creator if it enforces such a list.

Contract	Function	Threats

		Test coverage
		• Should:
		  • Claim & stake with stakeAmount
		  • == 0 → same effect as claim but with explicit unstaked logic.
		  • Claim & stake partial (stakeAmount < amount) → check veToken receives stakeAmount, user gets only liquidAmount.
		  • Claim & stake full (stakeAmount == amount) → no liquid/slashed amounts; all goes to veToken.
		• Should:
		  • Ensure totalClaimed += amount (full allocation), not just unstaked.
		  • Emit TokensClaimedAndStaked with correct values.
		
		Negative behavior
		• stakeAmount > amount → StakeAmountExceedsClaimAmount.
		• stakeAmount > 0 && unlockTime == 0 → UnlockTimeRequired.
		• stakeAmount > 0 && unlockTime < slashingEndTime → UnlockTimeBeforeSlashingPeriodEnds.
		• Any claim negative behavior:
		  • Before/after claim window, already claimed, invalid proof, amount == 0, paused.
		• Integration risk:
		  • If veToken.createLockFor reverts (e.g., not authorized creator, invalid unlock time format), entire claimAndStake reverts and no claim is recorded.

Contract	Function	Threats
		Negative tests • claimAndStake(amount, amount + 1, unlockTime, recipient, proof) → StakeAmountExceedsClaimAmount. • stakeAmount > 0 and unlockTime == 0 → UnlockTimeRequired. • stakeAmount > 0 and unlockTime < claimStartTime + slashingPeriodDuration → UnlockTimeBeforeSlashingPeriodEnds Use invalid Merkle proof → InvalidProof. • Double call by same address → AlreadyClaimed. • Call when paused → revert via whenNotPaused. • Mock veToken that reverts on createLockFor → ensure entire claimAndStake reverts (no partial side effects).
	revokeUnclaimedTokens(address recipient)	**Allows owner to withdraw remaining airdrop tokens after the claim window ends.** **Inputs** **recipient** • Control: Only owner can call function; owner controls this address. • Constraints: • Non-zero (InvalidTokenAddress).

Contract	Function	Threats
		Branches and code coverage **Intended branches** 1. After claim period ends • block.timestamp > claimDeadline required (ClaimPeriodNotEnded otherwise). • unclaimedRevoked must be false (AlreadyRevoked otherwise). • Marks unclaimedRevoked = true. • unclaimedAmount = token.balanceOf(this). • If unclaimedAmount > 0, transfers all to recipient and emits UnclaimedTokensRevoked. **Test coverage** • Should: • Revert if called before or at claimDeadline. • Revert if called twice (AlreadyRevoked). • On success, transfer full current token balance to recipient. • Emit UnclaimedTokensRevoked. **Negative behavior** • If contract has not been fully funded or some tokens were moved out earlier (via recoverTokens), unclaimedAmount may differ from totalAllocation - totalClaimed. That's by design, but must be understood. • Owner can choose recipient arbitrarily (could be a treasury or EOA).

Contract	Function	Threats
		Negative tests • Call before deadline → ClaimPeriodNotEnded. • Call twice → second call AlreadyRevoked. • Passing recipient = address(0) → InvalidTokenAddress.

AlmanakAirdropWithPreStake.sol	claimAndStake(uint256 amount, uint256 stakePercentageBps, uint256 unlockTime, address recipient, bytes32[] merkleProof)	Single entrypoint where user executes a pre-committed "claim + stake" defined in the Merkle tree.
		Inputs
		amount
		• Control: Fully controlled by caller (but must match Merkle tree).
		• Constraints:
		  • Must be > 0 (InvalidAmount).
		• Impact:
		  • Total allocation for the caller.
		  • Splits into staked and unstaked portions according to stakePercentageBps.
		stakePercentageBps
		• Control: Caller-supplied, but must match Merkle leaf.
		• Constraints:
		  • Must be > 0 and <= 10000 (InvalidStakePercentage). Merkle proof ensures this value is the same as encoded off-chain.
		• Impact:
		  • Determines how much of amount is staked:

Contract	Function	Threats

		stakeAmount = (amount * stakePercentageBps) / 10000;
		• Remaining part (amount - stakeAmount) is treated as unstaked and subject to slashing.
		
		unlockTime
		• Control: Caller-supplied, but must match Merkle leaf.
		• Constraints:
		  • unlockTime >= claimStartTime + slashingPeriodDuration (UnlockTimeBeforeSlashingPeriodEnds).
		  • Merkle leaf includes unlockTime, so off-chain commitment must match.
		  • Further constraints may be enforced inside veToken.createLockFor (e.g., week rounding, MAXTIME).
		• Impact:
		  • Lock end time for the staked portion in veToken.
		
		recipient
		• Control: Fully controlled by caller.
		• Constraints:
		  • Non-zero (InvalidTokenAddress).
		• Impact:
		  • Receives liquid (unstaked, post-slash) tokens.

Contract	Function	Threats

		merkleProof
		• Control: Provided by caller.
		• Constraints:
		  • Must satisfy:
		leaf = keccak256(
		bytes.concat(keccak256(abi.encode(msg.sender, amount, stakePercentageBps, unlockTime)))
		);
		  • MerkleProof.verify(merkleProof, merkleRoot, leaf) must be true, otherwise InvalidProof.
		• Impact:
		  • Binds caller to specific amount, stakePercentageBps, unlockTime triplet.
		
		Branches and code coverage
		Intended branches
		1. Valid pre-stake claim (happy path)
		  • Claim window open:
		  • block.timestamp >= claimStartTime
		  • block.timestamp <= claimDeadline
		  • Contract not paused (whenNotPaused).
		  • Caller has not claimed: hasClaimed[msg.sender] == false.

Contract	Function	Threats

		• Inputs:
		  • amount > 0 (§ InvalidAmount otherwise).
		  • stakePercentageBps > 0 && stakePercentageBps <= 10000.
		  • recipient != address(0). unlockTime >= claimStartTime + slashingPeriodDuration.
		• Merkle proof is valid (leaf matches).
		• State updates:
		  • hasClaimed[msg.sender] = true. stakeAmount = (amount * stakePercentageBps) / 10000.
		  • unstakedAmount = amount - stakeAmount.
		  • totalStaked += stakeAmount.
		  • slashingRate = calculateSlashingRate(block.timestamp).
		  • slashedAmount = (unstakedAmount * slashingRate) / 10000. liquidAmount = unstakedAmount - slashedAmount.
		  • totalClaimed += unstakedAmount (note: only the unstaked portion is counted in totalClaimed here).
		• Side effects:
		  • Approve and stake:

Contract	Function	Threats
		token.forceApprove(address(veToken), stakeAmount); **veToken.createLockFor(msg.sender, stakeAmount, unlockTime);** **token.forceApprove(address(veToken), 0);** • Transfer slashedAmount to slashingReceiver (if > 0). • Transfer liquidAmount to recipient (if > 0). • Emit TokensClaimedAndStaked. 2. Slashing disabled • If slashingPeriodDuration == 0, calculateSlashingRate() always returns 0 → slashedAmount == 0 and liquidAmount == unstakedAmount. 3. Full-stake scenario • stakePercentageBps == 10000 → stakeAmount == amount, unstakedAmount == 0. • slashedAmount == 0, liquidAmount == 0. • All tokens go to veToken lock. 4. Partial stake scenario • 0 < stakePercentageBps < 10000 → part staked, part slashed + liquid.

Contract	Function	Threats

		Test coverage (what to test)
		
		Happy path:
		• Merkle leaf with some (amount, stakePercentageBps, unlockTime):
		  • Check hasClaimed flips.
		  • Check totalStaked equals sum of staked portions.
		  • Check totalClaimed increments by unstakedAmount, not full amount.
		  • Check veToken receives stakeAmount and correctly records lock for msg.sender.
		  • Check recipient gets liquidAmount.
		  • Check slashingReceiver gets slashedAmount.
		
		Slashing extremes:
		• At claimStartTime → rate == initialSlashingRate, slashing is maximum.
		• After claimStartTime + slashingPeriodDuration → rate == 0, no slashing.
		
		Stake extremes:
		• Stake 100% (10000 bps) → liquidAmount == 0, slashedAmount == 0.
		• Stake small % and ensure math is correct.

Note to Users/Trust Assumptions

- **Centralization Risk**

The contracts rely on administrator-controlled parameters and token custody, meaning users must trust that the owner will not pause claiming, redirect slashed tokens, recover tokens from the contract prematurely, or revoke unclaimed allocations in a way that disadvantages participants.. For staking and lock-based flows, users must trust that authorized systems interacting with the escrow behave correctly. Slashed tokens flow to an owner-controlled address rather than being burned, and the owner can change this destination. Overall, correct operation depends on honest contract governance and the legitimacy of the published Merkle data.

Closing Summary

In this report, we have considered the security of Almanak. We performed our audit according to the procedure described above.

No critical issues in Almanak, just 2 issues of Low and Informational severity were found, which have been noted and resolved by the Almanak team.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

December 2025

For

ALMANAK

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com