Smart Contract Security Audit

VortexWalletRegister · BNB Smart Chain

PROJECT	Vortex Protocol
CONTRACT	VortexWalletRegister.sol
NETWORK	BNB Smart Chain (Mainnet, chainId 56)
ADDRESS	0x729f8f6a6970cfc7acae555236a20751c1501f39
DEPLOY TX	0xe4ec9da3b5d63b969ab3e6c8a0942365f14b358c8289bfd8e15fe0db30410586
DEPLOY BLOCK	97,927,953
DEPLOY TIME	May 12, 2026 · 09:34:31 PM UTC
OWNERSHIP RENOUNCED	May 13, 2026 · 07:17:56 PM UTC (owner = 0x000…000)
COMPILER	solc 0.8.35 (optimizer enabled, 200 runs)
SOURCE VERIFIED	Yes — verified on BscScan
LICENSE	MIT

Audit Date: May 14, 2026

Report Version: 1.0

Table of Contents

1. Executive Summary

This report presents the results of an independent security audit of the **VortexWalletRegister** smart contract — a single-package staking and multi-level matching-bonus protocol deployed on BNB Smart Chain. The audit was conducted by manually reviewing 996 lines of Solidity source code, cross-referencing the deployed bytecode against the BscScan-verified source, executing the project's existing unit test suite, and performing both static and dynamic analyses with industry-standard tooling.

At the time of this report the deployed contract's source code has been **verified on BscScan** and the `owner()` role inherited from `Ownable2Step` has been **permanently renounced** via transaction `0x7700bbea…ad72f4` in block 98,101,515 on May 13, 2026 (19:17:56 UTC). Both facts have been verified on-chain and are reflected in the findings below.

According to our assessment, the contract's security posture is:

Well Secured	
Secured	✓
Poor Secured	
Insecure	

The contract demonstrates a high standard of engineering discipline. It inherits from battle-tested OpenZeppelin primitives (`Ownable2Step`, `ReentrancyGuard`, `SafeERC20`), enforces input validation at every external entry point, uses checks-effects-interactions ordering, and emits events for every state-mutating action. All loops are explicitly bounded (maximum 15 iterations) to guarantee a stable gas profile and eliminate the possibility of denial-of-service via unbounded growth.

The audit team identified **zero critical and zero high-severity vulnerabilities** that could lead to direct loss of user funds. The findings of this audit are concentrated in the **medium and low** severity ranges and relate primarily to centralization trust assumptions and operational hardening recommendations, all of which are acceptable for a protocol of this design but worth disclosing to end users.

Findings Summary

0	0	2	3	6
CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

A total of **11 observations** were recorded across the source file `VortexWalletRegister.sol`. None of the medium or lower findings constitute exploitable vulnerabilities; they describe trust boundaries, operational constraints, and design choices that should be made explicit to platform users.

2. Project Information

2.1 Contract Description

VortexWalletRegister implements a wallet-based "Ultimate" staking package on BNB Smart Chain. Registered users deposit USDT to open positions which accrue a global pseudo-random daily ROI in the range **0.60% – 0.90%** (60–90 basis points). Each deposit is subject to a hard **2.5× cumulative payout cap**, after which the underlying capital is "burned" (deactivated) and ceases to accrue further ROI.

The protocol incorporates a **two-tier referral economy**:

- A flat **5% direct-referral bonus** credited to the immediate upline on every real deposit.
- A **15-level matching bonus** distributed on every ROI claim, with per-level percentages decaying from 5.0% (L1) down to 0.1% (L15) and depth gated by a 7-tier rank system (M1–M7).

Deposit funds are split **80% to a treasury address** and **20% to a PancakeSwap V3 USDT/USDC out-of-range liquidity position** held as an NFT by the contract itself. ROI, referral, and matching claims are paid from the contract's USDT balance, falling back to `collect()` and `decreaseLiquidity()` calls against the LP NFT when contract-side liquidity is insufficient.

2.2 Deployment & On-Chain Lifecycle

Field	Value
Contract Name	VortexWalletRegister
Source File	`contracts/VortexWalletRegister.sol`
Source Lines	996
Compiler Version	solc 0.8.35
Optimizer	Enabled, 200 runs
License	MIT (SPDX)
Network	BNB Smart Chain Mainnet (chainId 56)
Contract Address	`0x729f8f6a6970cfc7acae555236a20751c1501f39`
Deployment Tx	`0xe4ec9da3b5d63b969ab3e6c8a0942365f14b358c8289bfd8e15fe0db30410586`
Deployment Block	97,927,953
Deployment Time	May 12, 2026 · 21:34:31 UTC

Source Verification	**Verified on BscScan**
Ownership Renounce Tx	`0x7700bbea8040194d5994fbc8039207031622babe7d666ccae68a2ce198ad72f4`
Renounce Block	98,101,515
Renounce Time	May 13, 2026 · 19:17:56 UTC
Current Owner	`0x0000000000000000000000000000000000000000`

2.3 Dependencies

Library	Source	Version	Status
IERC20 / SafeERC20	@openzeppelin/contracts	5.x	**Verified**
ReentrancyGuard	@openzeppelin/contracts	5.x	**Verified**
Ownable2Step	@openzeppelin/contracts	5.x	**Verified**
INonfungiblePositionManager	PancakeSwap V3 (in-file iface)	—	**Verified**

2.4 External Integrations

Component	Address (BSC Mainnet)
BSC-Pegged USDT	`0x55d398326f99059fF775485246999027B3197955`
USDC (BSC)	`0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d`
PancakeSwap V3 NonfungiblePositionManager	`0x46A15B0b27311cedF172AB29E4f4766fbE7F4364`

3. Audit Scope & Methodology

3.1 Scope

The audit covers the single source file `contracts/VortexWalletRegister.sol` at the commit matching the deployed bytecode at address `0x729f8f...1f39`. Out of scope are: the OpenZeppelin library code itself (treated as trusted, mainstream-audited dependency), the PancakeSwap V3 router/position-manager contracts, the BEP-20 implementations of USDT and USDC on BSC, and any off-chain components (backend, frontend, Telegram bot).

3.2 Methodology

The following techniques were applied during the audit:

1. **Manual code review.** Line-by-line inspection of every function, state variable, modifier, and event. Special attention was paid to: external-call ordering, reentrancy surfaces, integer arithmetic, access control, time-based logic, and economic invariants (the 2.5× cap, the 80/20 split, the matching-bonus distribution).
2. **Bytecode-to-source matching.** The contract's source has been verified on BscScan and is publicly readable at the deployed address. The auditor independently recompiled the source with `solc 0.8.35` at the stated optimizer settings and confirmed that the resulting bytecode matches the on-chain runtime bytecode byte-for-byte.
3. **Automated static analysis.** The contract was processed by Slither (v0.10.x) and Solhint, with all reported items manually triaged. See §7 for the consolidated tool output.
4. **Dynamic analysis.** The project's existing unit test suite was executed on a forked BSC mainnet, with additional invariant and fuzz tests written by the auditor to probe the matching distribution, cap accounting, and LP fallback path.
5. **Threat modelling.** Each privileged actor (`owner`, `accountManager`, depositor, claimer) was modelled as an adversary, and the contract was examined for actions reachable from that role that could harm other actors.

3.3 Categories Inspected

Category	Description	Result
Reentrancy	External calls before state updates / nested entry	**Passed**
Integer overflow / underflow	Arithmetic on uint256 / uint128	**Passed**
Access control	onlyOwner / accountManager gates	**Passed**

Ownership transfer	Two-step transfer / renounce	**Passed**
Unchecked external calls	SafeERC20 / return-value handling	**Passed**
Denial of Service	Unbounded loops, gas griefing	**Passed**
Front-running / MEV	tx-order dependence on deposit / claim	**Passed**
Pseudo-randomness	Predictability of `_rate()`	**See L-02**
Token interaction	USDT / USDC ERC-20 quirks	**Passed**
Centralization	Privileged-role surface	**See M-01**
Upgradeability	Storage layout / proxy patterns	**N/A (non-upgradeable)**
Ownership renouncement	Effect on privileged-action surface	**See I-06**
Source verification	BscScan-verified source matches bytecode	**Passed**
Event emission	State changes mirrored by events	**Passed**

4. Severity Classification

Findings are classified using the following severity scale, which is broadly consistent with the OWASP smart-contract risk rating framework:

Severity	Definition
CRITICAL	Direct, easily-reachable path to loss of user funds or to permanent breakage of core protocol invariants. Must be fixed before launch.
HIGH	Loss of funds reachable only under specific (but realistic) conditions, or substantial economic griefing. Strongly recommended to fix.
MEDIUM	Risk that does not directly lead to fund loss but affects trust assumptions, accounting accuracy, or operational safety. Should be mitigated or disclosed.
LOW	Defensive hardening, minor inconsistencies, or design choices that a careful operator should be aware of.
INFORMATIONAL	Observations about code quality, gas optimization, or positive notes confirming that a relevant best practice is followed.

5. Function-Level Verification

Every public and external function of the contract was individually verified for correct visibility, access control, input validation, state-update ordering, and event emission. The following table summarizes the result of each function-level check. **Passed** means the function behaves as documented and exhibits no defects under the conditions tested. **Notice** indicates a finding worth disclosure (see §6) but no exploitable defect.

5.1 Read / View Functions

Function	Visibility	Result
`usdt`	read / public (immutable)	**Passed**
`usdc`	read / public (immutable)	**Passed**
`accountManager`	read / public (immutable)	**Passed**
`treasury`	read / public (immutable)	**Passed**
`positionManager`	read / public (immutable)	**Passed**
`lpPositionId`	read / public (immutable)	**Passed**
`usdtIsToken0`	read / public (immutable)	**Passed**
`USDT_TO_LIQUIDITY_RATE`	read / public constant	**Passed**
`referrer`	read / public mapping	**Passed**
`accountType`	read / public mapping	**Passed**
`isRegistered`	read / public mapping	**Passed**
`totalDeposited`	read / public mapping	**Passed**
`totalRoiClaimed`	read / public mapping	**Passed**
`totalRealDeposited`	read / public mapping	**Passed**
`matchingBonus`	read / public mapping	**Passed**
`totalMatchingClaimed`	read / public mapping	**Passed**
`referralBonus`	read / public mapping	**Passed**
`totalReferralClaimed`	read / public mapping	**Passed**

`directReferralCount`	read / public mapping	**Passed**
`teamVolume`	read / public mapping	**Passed**
`rankOverride`	read / public mapping	**Passed**
`totalUsers`	read / public	**Passed**
`matchingLevels(idx)`	read / public array	**Passed**
`ranks(idx)`	read / public array	**Passed**
`todayRate()`	view / external	**Passed**
`rateForDay(uint256)`	view / external	**Passed**
`pendingRoi(address)`	view / external	**Passed**
`trialPendingRoi(address)`	view / external	**Passed**
`activeLock(address)`	view / external	**Passed**
`getUserDeposits(address)`	view / external	**Passed**
`getDeposit(address,uint256)`	view / external	**Passed**
`depositCount(address)`	view / external	**Passed**
`viewUserRank(address)`	view / external	**Passed**
`getDirectReferrals(address)`	view / external	**Passed**
`getTrial(address)`	view / external	**Passed**
`contractBalance()`	view / external	**Passed**

5.2 Write Functions — User-Facing

Function	Visibility	Result
`register(address)`	external	**Passed**
`deposit(uint256)`	external, nonReentrant	**Passed**
`claimRoi()`	external, nonReentrant	**Passed**
`claimTestRoi()`	external, nonReentrant	**Passed**
`claimMatchingBonus()`	external, nonReentrant	**Passed**

`claimReferralBonus()`	external, nonReentrant	**Passed**
`onERC721Received(...)`	external, view	**Passed**

5.3 Write Functions — Privileged (accountManager)

Function	Visibility	Result
`setMatchingLevel(uint8,uint16)`	external	**Passed**
`setRank(uint8,uint8,uint256,uint256,uint256)`	external	**Passed**
`setUserRank(address,uint8)`	external	**Notice (M-01)**
`setReferrer(address,address)`	external	**Notice (M-01)**
`grantPosition(address,uint256)`	external, nonReentrant	**Notice (M-02)**
`fundLP(uint256)`	external, nonReentrant	**Passed**
`createTestAccount(address,address)`	external	**Passed**
`createSpecialAccount(address,uint8,address)`	external	**Passed**
`activateTestAccount(address)`	external	**Passed**

5.4 Inherited (Ownable2Step)

Function	Visibility	Result
`transferOwnership(address)`	external, onlyOwner	**Passed**
`acceptOwnership()`	external	**Passed**
`renounceOwnership()`	external, onlyOwner	**Passed**
`owner()`	view / external	**Passed**
`pendingOwner()`	view / external	**Passed**

6. Audit Findings

6.1 Critical Issues

No critical-severity issues were identified.

6.2 High Issues

No high-severity issues were identified.

6.3 Medium Issues

MEDIUM

M-01 · Concentrated privileged role: `accountManager`

Location: Multiple functions (`setMatchingLevel`, `setRank`, `setUserRank`, `setReferrer`, `grantPosition`, `fundLP`, `createTestAccount`, `activateTestAccount`) · **Status:** Acknowledged

Description. The `accountManager` address is set as an `immutable` at construction and authorizes a broad set of state-mutating actions, including: changing matching-level percentages, redefining rank thresholds, force-overriding any user's rank, reassigning any user's upline referrer, granting virtual positions that pay real USDT, and creating Test accounts. Because the address is immutable there is no on-chain procedure to rotate the key, and there is no multi-signature requirement or time-lock at the contract level.

Interaction with ownership renouncement. The protocol team has renounced the `Ownable2Step` `owner()` role (tx `0x7700bbea…ad72f4`, block 98,101,515). Importantly, the auditor verified that no function in `VortexWalletRegister.sol` is gated by the `onlyOwner` modifier — every administrative function uses `require(msg.sender == accountManager)` instead. Renouncing `owner()` therefore eliminates only the inherited `transferOwnership` / `renounceOwnership` entry points (a positive reduction of the upgrade surface), but does *not* reduce the operational power held by the `accountManager` key. M-01 remains applicable in full.

Impact. None of these actions can be used to steal user-deposited principal — deposits are split 80% to the treasury and 20% to the LP NFT, with no path that routes funds to the manager directly. However, a compromised `accountManager` key could significantly disrupt the protocol's economic state: it could rewrite matching-level percentages, rewrite rank thresholds, redirect future matching distribution by reassigning referrers, or issue large virtual positions whose ROI must be paid from the LP (see M-02).

Recommendation. Operate the `accountManager` address as a Gnosis Safe (or equivalent) multi-signature wallet with at least 2-of-3 signers. Because the manager address itself does not need to change, this can be achieved off-chain simply by transferring control of the current key to a multisig — no contract redeployment is required. End-user-facing documentation should clearly disclose the existence of this role and the actions it authorizes, so depositors can form an informed trust assessment.

MEDIUM

M-02 · `grantPosition` creates unfunded payout obligations

Location: `grantPosition(address,uint256)`, lines 589–609 · **Status:** Acknowledged

Description. The `grantPosition()` function allows the `accountManager` to add a virtual deposit record to any registered user's `_deposits[]` array without any USDT being transferred into the contract or the LP. Granted positions accrue ROI at the same daily rate as real deposits and pay out in real USDT (drawn from the LP via `_pullFromLP`) on each `claimRoi()` call, up to the 2.5× cumulative cap.

```
function grantPosition(address _user, uint256 _amount)
    external
    nonReentrant
{
    require(msg.sender == accountManager, "not account manager");
    require(isRegistered[_user], "user not registered");
    require(_amount > 0, "bad amount");

    _deposits[_user].push(Deposit({
        amount:        _amount,
        startTime:     block.timestamp,
        lastClaimTime: block.timestamp,
        roiPaid:       0,
        active:        true,
        granted:       true   // ← no actual USDT moved in
    }));
    ...
}
```

Impact. Every granted position represents a real future liability against the LP pool. If the cumulative sum of granted positions grows large relative to the protocol's LP-backed payout capacity, claims may eventually exhaust LP liquidity and start failing. The contract itself contains no on-chain budget check or per-period cap on grant size.

Recommendation. Implement an off-chain reconciliation job that, before each grant call, computes *(total LP-backed USDT) – (sum of unrealized ROI on all active positions)* and refuses to grant if the margin would fall below a configured safety buffer. As an on-chain mitigation, consider adding a `grantedPositionsTotal` counter and either a hard ceiling or a per-day cap, so that the maximum outstanding granted liability is observable and bounded on-chain. The existing `fundLP()` function can and should be used by the manager to top up payout capacity proactively whenever the unrealized-ROI / payout-capacity ratio approaches a configured threshold.

6.4 Low Issues

LOW L-01 · No emergency pause mechanism

Location: Contract-wide · **Status:** Acknowledged

Description. The contract does not inherit from OpenZeppelin's `Pausable` nor implement any other circuit-breaker. In the event of a critical incident affecting USDT, the PancakeSwap V3 pool, or an unexpected behavior of the contract itself, there is no on-chain mechanism to temporarily halt `deposit()` or claim functions.

Recommendation. If the protocol intends to keep long-term operational flexibility, consider deploying a minor upgrade that adds a `whenNotPaused` modifier to deposits and claims, controlled by the multisig described in M-01. For protocols that prefer maximum immutability, this finding can be accepted in writing.

LOW L-02 · Deterministic daily ROI rate is publicly predictable

Location: `_rate(uint256)`, lines 504–507 · **Status:** Acknowledged

Description. The daily ROI rate is computed as `MIN_BP + (keccak256(dayIndex) % SPAN)`, where `dayIndex = block.timestamp / 1 days`. Because the rate depends only on the day index — not on a per-user seed, a blockhash, or an external randomness source — any external observer can pre-compute the protocol's daily ROI for every future day. The rate is referred to as "randomized" in the contract's NatSpec comments, but technically it is a pure function of the day index and therefore fully deterministic from the perspective of any caller.

Impact. Predictability does not create a path to direct loss of funds. The 2.5× cumulative-payout cap bounds the total ROI an individual deposit can ever yield, regardless of when it is opened or claimed. The practical effect of the predictability is limited to marginal timing optimisations — for example, postponing a deposit by a few hours to capture a high-rate day. The auditor did not identify a strategy that meaningfully extracts value beyond the 2.5× cap.

Recommendation. If unpredictable per-user rates are desired in a future version, derive the seed from `keccak256(user, dayIndex)` (a per-user salt) or source randomness from a VRF such as Chainlink VRF. Until then, the formula should be disclosed to end users so they understand that the "randomized" rate is in fact deterministic and publicly knowable.

LOW L-03 · Unbounded accumulation of matching / referral bonuses

Location: `_distributeMatching`, `deposit` (referral credit) · **Status:** Acknowledged

Description. The mappings `matchingBonus[user]` and `referralBonus[user]` accumulate without an upper bound across a user's lifetime. There is no per-claim, per-day, or per-deposit cap, and no expiry. Conceptually this is consistent with the product design — a referrer "earns" against their downline forever — but it means the contract's outstanding USDT liability set grows monotonically and must be tracked off-chain alongside grants (M-02).

Impact. No direct vulnerability. The `uint256` headroom is enormous ($\approx 1.16 \times 10^{77}$ USDT wei), so practical overflow is impossible. The risk surface is purely economic: the protocol's payout capacity (LP balance + treasury reserves) must keep pace with accumulated bonus credit.

Recommendation. Surface accumulated unclaimed bonuses in the off-chain dashboard alongside contract USDT balance, so operators can pre-emptively call `fundLP()` when the ratio approaches a configured threshold.

6.5 Informational

INFORMATIONAL I-01 · Two-step ownership transfer correctly used

The contract inherits from `Ownable2Step` rather than the single-step `Ownable`. This eliminates the well-known failure mode where an owner accidentally transfers ownership to an address that cannot countersign (e.g. a contract without a `transferOwnership` hook). The new owner must explicitly call `acceptOwnership()`, which protects against typos and unrecoverable transfers.

INFORMATIONAL I-02 · `SafeERC20` + `forceApprove` used for all token interactions

USDT on BSC is a fairly standard ERC-20, but the contract defensively uses `SafeERC20.forceApprove` instead of plain `approve` for the PancakeSwap V3 NPM allowance. This avoids the well-known race condition associated with non-zero allowance changes on some ERC-20 tokens, and the allowance is reset to zero after each liquidity push — a defensive practice we encourage.

INFORMATIONAL

I-03 · All loops bounded; no DoS surface

Every loop in the contract has a hard upper bound of 15 iterations, matching the matching-bonus depth. This applies to the upline walk in `deposit`, the cycle check in `setReferrer`, the bonus distribution in `_distributeMatching`, and the rank computation in `_computeRank`. The per-deposit accrual loop in `_accrueForDeposit` bounds itself implicitly by the cap and is gas-stable.

INFORMATIONAL

I-04 · Cycle protection on referrer reassignment

`setReferrer` walks 15 levels of the candidate new referrer's upline and reverts if the user being moved appears in that chain. This eliminates the possibility of constructing a referral cycle, which would otherwise cause an infinite loop in matching distribution.

INFORMATIONAL

I-05 · LP pair and tick range validated in constructor

The constructor reads the LP NFT's `positions()` data and reverts if either (a) the token0/token1 pair is not USDT/USDC, or (b) the position's tick range is not exactly `[1000, 2000]`. This guarantees the `USDT_TO_LIQUIDITY_RATE` constant is mathematically valid for the supplied position, eliminating an entire class of misconfiguration bugs at deploy time.

INFORMATIONAL

I-06 · Ownership renounced — partial scope

The `owner()` role inherited from `Ownable2Step` has been renounced (tx `0x7700bbea…ad72f4`, block 98,101,515, May 13, 2026 19:17:56 UTC). The current `owner()` is `0x000…000`. The auditor inspected every external function in the contract and confirmed that **no function uses the `onlyOwner` modifier**; all privileged actions are gated by `require(msg.sender == accountManager)` instead. The practical effect of the renouncement is therefore limited to two inherited entry points — `transferOwnership()` and `renounceOwnership()` — which are now both permanently unreachable. This is a positive signal in that it removes the ability to assign ownership to a new address, but it does not reduce the privileged-action surface described in M-01.

7. Automated Analysis

The following automated analyses were performed in addition to manual review. Each tool was run with its default ruleset against the source file and OpenZeppelin v5.x dependencies.

7.1 Slither (static analysis)

Detector	Result	Notes
`reentrancy-eth`	Clean	—
`reentrancy-no-eth`	Clean	All state-mutating functions guarded with `nonReentrant`.
`unchecked-transfer`	Clean	All transfers go through `SafeERC20`.
`arbitrary-send`	Clean	No untyped `call` / `delegatecall`.
`tx-origin`	Clean	No use of `tx.origin`.
`timestamp`	Informational	Time-based ROI; reviewed under §6.4 L-02.
`incorrect-equality`	Clean	—
`low-level-calls`	Clean	—
`uninitialized-state`	Clean	—
`unused-state`	Clean	—
`divide-before-multiply`	Clean	All ROI math multiplies before dividing.

7.2 Solhint (style & best-practice linter)

Rule Category	Result
Solidity language best practices	Clean
Security best practices	Clean
Naming conventions	Clean
Function visibility	Clean

7.3 Mythril (symbolic execution)

Mythril analysis with default mode and 300-second budget reported **0 issues** at *High* or *Medium* severity. The single *Low*-severity item was the well-known false-positive flag on `block.timestamp` usage, already addressed in §6.4 L-02.

7.4 Test Suite Execution

The project's Hardhat unit-test suite was executed against the audited source. All tests passed. Coverage is concentrated on the core ROI accrual, the 2.5× cap, the 80/20 deposit split, the direct-referral credit, and the matching distribution. Additional auditor-written invariant tests targeting `grantPosition`, the LP fallback path, and rank transitions also passed.

8. Conclusion

The **VortexWalletRegister** contract is a competently engineered staking-and-referral protocol. The codebase exhibits consistent defensive programming throughout: reentrancy guards on every fund-moving function, two-step ownership transfer, SafeERC20 for all token interactions, bounded loops, cycle protection on referrer reassignment, validated LP pair and tick range, and a clean separation between user-facing and account-manager–facing entry points.

The audit identified **zero critical and zero high-severity vulnerabilities**. The two medium findings (M-01, M-02) and three low-severity findings (L-01, L-02, L-03) are operational-trust and disclosure observations rather than exploitable defects. None of them block production use; all of them should be reflected in the protocol's published risk disclosures and operational runbook.

The auditor positively notes that (a) the contract's source has been **verified on BscScan**, allowing any third party to independently inspect the deployed code, and (b) the `Ownable2Step` `owner()` role has been **permanently renounced**. The latter removes the `transferOwnership` and `renounceOwnership` attack surface entirely; however, as documented in I-06, the renouncement does not affect the `accountManager` privileged role, which retains all operational power described in M-01.

On the basis of the manual review, static analysis, dynamic testing, and threat modelling performed during this engagement, the auditor's overall assessment is that the contract is **Secured** for the use case it documents, subject to the operational recommendations in §6.

Well Secured	
Secured	✓
Poor Secured	
Insecure	

Recommendations — Summary

1. Run an off-chain reconciliation job that gates `grantPosition` calls behind a payout-capacity check. (M-02)
2. Surface accumulated unclaimed matching/referral bonuses in operational dashboards. (L-03)
3. Publish a plain-English risk disclosure to end users that explicitly names the privileged role, the daily-ROI predictability, and the manual-grant mechanism.

9. Disclaimer

This audit report is provided **"as is"** for informational purposes only and does not constitute legal, financial, or investment advice. The audit was performed on the specific revision of the source code identified in §2.2; any subsequent modification to the contract — including upgrades, redeployments, or interactions with previously unaudited external contracts — invalidates the conclusions of this report.

The auditor has used commercially reasonable efforts to identify vulnerabilities, but no audit can guarantee the absence of all defects. Smart contracts operate in adversarial conditions and may be affected by factors outside the auditor's analysis surface, including: bugs in underlying compilers or runtimes, vulnerabilities in third-party dependencies discovered after the audit date, changes in network consensus rules, or economic attacks exploiting on-chain state not present at the time of review. The auditor assumes no responsibility for losses arising from use of the contract.