ZENFI

SMART CONTRACT SECURITY AUDIT

Referral Staking Security Audit

Independent security assessment of ZenReferralStaking — the NFT-key-bound tiered $ZEN staking contract with a 3-line referral share, deployed on BNB Smart Chain with all administrative roles renounced.

Engagement	Security Audit
Commit	e6446d6dce44b7a45eec52686662379586bd537c
Branch	feat/academy
Report date	2026-07-31
Result	0 Critical · 1 Major · admin roles renounced on-chain

Table of Contents

Project Summary

PROJECT	ZenFi — ZenReferralStaking
CONTRACTS UNDER REVIEW	`ZenReferralStaking.sol`
TYPE	Tiered $ZEN staking bound to an NFT key (by `nftId`, not by wallet), with an NFT-level APR boost and a 3-line referral share
PLATFORM / CHAIN	EVM — BNB Smart Chain Mainnet (chainId 56)
LANGUAGE	Solidity 0.8.28
COMPILER	solc 0.8.28 — optimizer enabled, runs = 200, no viaIR
DEPENDENCIES	OpenZeppelin Contracts v5.6.1 (AccessControl, ReentrancyGuard, Pausable, SafeERC20)
CODEBASE COMMIT	e6446d6dce44b7a45eec52686662379586bd537c
BRANCH	`feat/academy`
AUDIT METHOD	Multi-reviewer manual review + adversarial verification, anchored to the live chain state: 7 per-dimension finder passes (access control & the renouncement, reward-pool solvency, the referral leg, accrual & withdrawal maths, external-call safety & token integration, key-ownership binding & the core integration, DoS / gas / input validation), each followed by an independent refutation pass; every load-bearing claim was then re-verified directly against BNB Smart Chain — including a full replay of the contract's complete `RoleGranted` / `RoleRevoked` history — and against the project's own test suite.
REPORT DATE	2026-07-31

Audit Scope

FILE	SHA-256	DESCRIPTION
contracts/staking/ZenReferralStaking.sol	bb8d7f9b37b578947a6df056d41ae1f7e81fe6927dc96618037a84296411eff1	Tiered $ZEN staking keyed by NFT-key `nftId` (never by wallet): four lock tiers with per-tier APR, an NFT-level APR boost on the 1-year tier only, a snapshot-and-re-snapshot rate model, a 3-line referral share paid on top of every claim, and a core-triggered `claimFor` hook on level-up.

Out of scope: the OpenZeppelin library code; the `ZenFiNftKey` marketing core and the `$ZEN` (`TaxedToken`) token, both covered by the separate **ZenFi Protocol** report and consulted here only through the interfaces this contract calls (`ownerOf` / `levelOf` / `nftIdOf` / `getUplineChain` / `treasury`, and the ERC-20 surface); the ZenFi front end, backend and indexer; and the **economic sustainability** of the advertised APR schedule, which is a business question rather than a code-security one. The SHA-256 hash is of the source as deployed.

Deployed Addresses & On-Chain Verification

The audited `ZenReferralStaking` source — whose SHA-256 hash is listed in **Audit Scope** above — is deployed on **BNB Smart Chain (mainnet, chainId 56)** at the address below, at block 106,283,863 (2026-06-25 11:17 UTC). To confirm the live code matches this report, open it on BscScan, check that the source is **Verified**, and compare it against the Scope hash and the repository at commit `e6446d6d`.

Network: **BNB Smart Chain — mainnet (chainId 56)**

CONTRACT	SOURCE	ADDRESS (BSCSCAN)	SCOPE
ZenReferralStaking	contracts/staking/ZenReferralStaking.sol	0x1b7418DA30b5C9eD15C228A992ba3d4a1cEE8B3C	In scope (audited)
$ZEN — TaxedToken (staking token)	contracts/tokens/Zen.sol	0xCF361DE24b8fe44cCF1E2E51A28eE124174f23DF	Dependency — audited separately
ZenFiNftKey (marketing core)	contracts/ZenFiNftKey.sol	0x74af0f878368e039eb0652C8524Bb8B34a01731C	Dependency — audited separately

Administrative control has been renounced on-chain, and this audit verified it directly. Every `RoleGranted` / `RoleRevoked` event ever emitted by the contract was replayed from deployment to block 113,225,000 — **nine events in total**, the full history. `ADMIN_ROLE` was revoked from the deployer at deployment and from the admin account in tx `0xe2255de1d57393481374d79e8ae9f7884a33dc5b08ba947823a9c9d56f11eb5c` (block 113,221,088, 2026-07-31 14:52 UTC); `DEFAULT_ADMIN_ROLE` was revoked in tx `0xa9a3338c9d9c38925011d8de1832b3c12cf0b5203524bdc374319158a02fb683` (block 113,221,152, 2026-07-31 14:53 UTC). **No account holds `ADMIN_ROLE` or `DEFAULT_ADMIN_ROLE`.** Because `DEFAULT_ADMIN_ROLE` is the role-admin of every role and the contract exposes no other grant path, no role can ever be granted again. `CORE_ROLE` remains held by exactly one address — the `ZenFiNftKey` core contract — which is required for the level-up `claimFor` hook and cannot call anything else. The live configuration read at block ~113,225,000 is therefore permanent: `baseRewardRates = [30, 40, 50, 100]` %, `lockPeriods = [30, 90, 180, 365]` days, `earlyWithdrawalPenalties = [3000, 4500, 6000, 7500]` bps, `minStake = 1 ZEN`, `maxStake = 1,000,000 ZEN` per key, `referralPercentBps = 500`, `isEarlyWithdrawalEnabled = false`, `paused = false`, `stakingToken = $ZEN`, `core = ZenFiNftKey`.

Audit Methodology

`ZenReferralStaking` was reviewed with manual code review and tool-assisted analysis organised as seven independent specialist passes: access control, roles and the consequences of the renouncement; reward-pool solvency and the commingling of principal with the reward budget; the referral payout leg and its dependence on the marketing core; interest accrual, the rate-snapshot model and the withdrawal maths; external-call safety, reentrancy, CEI ordering and token integration; NFT-key binding, transfer semantics and the core level-up integration; and denial of service, unbounded growth, gas and input validation.

Each candidate issue was then re-examined by an independent adversarial pass instructed to **refute** it — checking every cited line, every arithmetic claim and every stated consequence against the source, and discarding anything not reproducible, already prevented by a guard the finder missed, or neutralised by the renounced roles. Surviving findings were deduplicated into the list below and every severity was re-rated against the **live deployed configuration**.

Because the contract is deployed and immutable, this report is anchored to on-chain fact rather than to a working tree. The complete `RoleGranted` / `RoleRevoked` history was replayed from the deployment block to establish that no privileged account remains; every configuration value quoted in this report was read directly from the contract; the staked principal and the reward balance were measured on-chain; and the project's own test suite was executed against the deployed source (**16 passing, 4 failing** — see ZS-10, a stale specification rather than a code defect).

Executive Summary

`ZenReferralStaking` is ZenFi's $ZEN staking contract. Its distinguishing property is that a position belongs to an **NFT key**, not to a wallet: stakes are stored under `nftId` and the actor is resolved live through `core.ownerOf(nftId)`, so selling or transferring the key carries the position — principal and rewards — to the new owner. It offers four lock tiers (30 / 90 / 180 / 365 days at 30 / 40 / 50 / 100 % annual), an NFT-level APR boost of +2 percentage points per level that applies **only** to the 1-year tier, and a referral share of 5% of every claim paid to each of up to three sponsor lines, on top of the claim and out of the pool. It is built on OpenZeppelin v5.6.1 under Solidity 0.8.28.

The engineering is sound and the code contains no externally-exploitable vulnerability. Checks-effects-interactions ordering is correct on every value-moving path, `nonReentrant` sits on all four of them, arithmetic is checked and was verified unit-by-unit against the live bounds with no reachable overflow or dust-truncation, authorisation is a single consistent `ownerOf` check, and the token integration is safe for `$ZEN` (which taxes only DEX buys and sells, and excludes this contract from fees regardless). No untrusted party can steal, mint or drain — there are **no Critical findings**.

Administrative control has been renounced, and this audit verified it against the full on-chain role history. No account holds `ADMIN_ROLE` or `DEFAULT_ADMIN_ROLE`; since the latter is the role-admin of every role and the contract has no other grant path, no role can ever be granted again. All nine privileged functions — `withdrawTokens`, `setStakingToken`, `setBaseRewardRates`, `setReferralPercent`, `setEarlyWithdrawalPenalties`, `toggleEarlyWithdrawal`, `depositTokens`, `pause`, `unpause` — are permanently uncallable. **This closes the entire centralization surface of the contract**, including the two most serious pre-existing risks: an admin could have transferred the whole balance out (staked principal included) with no reserve check and no event (ZS-06), and could have re-denominated every open position into a different token (ZS-07). Both are now impossible for anyone, forever.

In total **22 findings** were confirmed after deduplication: **0 Critical, 1 Major, 6 Medium, 8 Minor and 7 Informational. 3 are resolved by the renouncement itself** (ZS-06, ZS-07, ZS-08 — the admin-drain, token-repoint and unbounded-parameter surfaces); the other 19 are properties of an immutable contract and are recorded as *Acknowledged*, each with the off-chain control that manages it.

The single **Major** finding is not a code defect but a **funding obligation**: staked principal and the reward budget share one unreserved token balance, the contract holds no solvency invariant and no global stake cap, and the renouncement removed every lever that could have throttled liabilities. At the live parameters the top tier costs the pool up to **131% of principal per year** (114% APR plus the 15% referral surcharge). Measured on-chain at the time of writing the contract holds **1,215,921 ZEN** against **223,387 ZEN** of open principal across 58 keys — a free reward buffer of **992,534 ZEN**, which covers even a worst-case all-top-tier reading of the current book for roughly **3.4 years**. The pool is comfortably solvent today; what is missing is an on-chain guarantee that it stays that way as TVL grows. The mitigation survives the renouncement intact: **anyone can top the pool up with a plain ERC-20 transfer of $ZEN to the contract** — `depositTokens` was only an event-emitting wrapper and no payout path consults an internal ledger.

> **Audit result:** `ZenReferralStaking` is a compact, competently-engineered staking contract with **no Critical findings and no externally-exploitable path to user funds**. Its entire administrative surface has been **renounced on-chain and verified here against the complete role history**, which permanently removes the admin-drain and token-repoint risks that dominate this contract class. The residual risk is **economic, not technical**: rewards and referrals are paid from the same balance that holds staked principal, with no on-chain solvency invariant and — now — no parameter left to adjust. That makes a published, funded top-up policy and a public solvency figure the load-bearing controls for this product.

Trust Model — Capability Summary

The matrix below consolidates — directly from the findings — what each actor can and cannot do against user funds under the as-deployed, post-renouncement configuration. The throughline: **no party, privileged or otherwise, can take a user's stake**; the residual risk is whether the reward pool stays funded as TVL grows.

ACTOR	WHAT THEY CAN / CANNOT DO TO USER FUNDS
Untrusted third party / random caller	**Cannot** stake on someone else's key, claim someone else's reward, or withdraw someone else's principal. Every value-moving function resolves authorisation through `core.ownerOf(nftId) == msg.sender`. No reentrancy, arithmetic or validation path was found that changes this — **0 Critical, 0 Major externally-exploitable findings.**
Former admin (roles renounced)	**Cannot do anything.** `ADMIN_ROLE` and `DEFAULT_ADMIN_ROLE` have no holder and can never be granted again (verified across the contract's complete 9-event role history). The pool cannot be drained (ZS-06), the staking token cannot be repointed (ZS-07), rates and penalties cannot be rewritten (ZS-08), and the contract cannot be paused. **The centralization surface is closed permanently.**
Staker	Owns the position for as long as they own the key. Rewards are claimable from the first second; **principal is locked for the full tier term** (up to 365 days) with **no early exit at any price** — `isEarlyWithdrawalEnabled` is false and can never be switched on (ZS-11). A claim requires the pool to hold 115% of the reward, not 100% (ZS-03), and `withdraw` pays principal and reward as one all-or-nothing transfer (ZS-02).
Key holder / buyer of a key	The staked balance is a **bearer instrument attached to the NFT key**: whoever holds the key controls the principal and the accrued reward. An ERC-721 approval on the key is therefore an approval over the staked $ZEN, and an accrued-but-unclaimed reward settles to whoever owns the key at claim time (ZS-14). This is the contract's documented design, not a defect — but it must be surfaced to users.

ACTOR	WHAT THEY CAN / CANNOT DO TO USER FUNDS
The reward pool (the residual risk)	Principal and rewards share **one unreserved balance** with no solvency invariant and no global stake cap (ZS-01). Today: 1,215,921 ZEN held against 223,387 ZEN of open principal. If liabilities ever outrun the buffer, claims and withdrawals fail in first-come-first-served order and **no on-chain party can intervene.** The one control that survives the renouncement is a **permissionless top-up by plain ERC-20 transfer.**
The `ZenFiNftKey` core (still privileged)	This contract reads eligibility, APR, the payee and the sponsor chain live from the core, and holds no independent copy. The core's own `DEFAULT_ADMIN_ROLE` / `ADMIN_ROLE` are **not** renounced (verified on-chain), so staking economics remain downstream of the core's administration (ZS-05). `CORE_ROLE` here grants only `claimFor`, which pays a user their own rewards and can move nothing else.

This matrix is a **code-security** view of the on-chain contract. The **economic sustainability** of the advertised APR schedule, the funding of the reward pool, and any legal or regulatory classification of the referral structure are **out of scope** and were neither assessed nor endorsed. Readers should perform their own due diligence, consistent with the Disclaimer.

Findings Summary

The table below enumerates all confirmed findings after merging duplicates surfaced independently by multiple reviewers. 22 findings in total: 0 Critical, 1 Major, 6 Medium, 8 Minor, 7 Informational. **3 of 22 are resolved (status Fixed)** in the reviewed working tree; the remainder are documented design/centralization acknowledgements.

0	1	6	8	7
CRITICAL	MAJOR	MEDIUM	MINOR	INFORMATIONAL

ID	TITLE	CONTRACT	SEVERITY	CATEGORY	STATUS
ZS-01	Staked principal and the reward budget share one unreserved balance: no solvency invariant, no global stake cap, and — after the renouncement — no lever left to throttle liabilities	ZenReferralStaking.sol	MAJOR	Reward-pool solvency	ACKNOWLEDGED
ZS-02	`withdraw` pays principal and reward in one all-or-nothing transfer, and there is no principal-only or partial exit	ZenReferralStaking.sol	MEDIUM	Logical issue / fund safety	ACKNOWLEDGED
ZS-03	The referral leg is not fail-soft: a claim reverts unless the pool holds 115% of the reward, and the core's `try/catch` turns that into a silent no-op	ZenReferralStaking.sol	MEDIUM	Reward-pool solvency / denial of service	ACKNOWLEDGED

ID	TITLE	CONTRACT	SEVERITY	CATEGORY	STATUS
ZS-04	`_payReferral` applies no sponsor qualification, so a staker can capture the whole 15% surcharge with a three-key upline of their own	ZenReferralStaking.sol	MEDIUM	Logical issue / economics	ACKNOWLEDGED
ZS-05	The renouncement does not extend to `ZenFiNftKey` : eligibility, APR, the payee and the sponsor chain are all live reads from a core whose administration is still privileged	ZenReferralStaking.sol	MEDIUM	Access control / cross-contract trust	ACKNOWLEDGED
ZS-06	`withdrawTokens` allowed the admin to transfer the entire balance — staked principal included — with no reserve check and no event	ZenReferralStaking.sol	MEDIUM	Access control / centralization	FIXED
ZS-07	`setStakingToken` could re-denominate every open position while the principal stayed in the old token	ZenReferralStaking.sol	MEDIUM	Access control / centralization	FIXED
ZS-08	Unbounded rate, referral and penalty setters could have rewritten the economics of open positions — including a rate that bricks a position through checked-arithmetic overflow	ZenReferralStaking.sol	MINOR	Access control / insufficient validation	FIXED
ZS-09	The per-key `stakes` array is append-only and never compacted, so `claimAllRewards` and the core's level-up hook grow without bound	ZenReferralStaking.sol	MINOR	Denial of service / gas	ACKNOWLEDGED
ZS-10	`levelBoost` contradicts its own NatSpec and 4 of the 20 shipped tests: level 5 earns +0%, not the documented +2%, and level 12 earns +14%, not +16%	ZenReferralStaking.sol	MINOR	Code quality / documentation	ACKNOWLEDGED

ID	TITLE	CONTRACT	SEVERITY	CATEGORY	STATUS
ZS-11	`Pausable` is permanently inert and early withdrawal is permanently disabled: no circuit breaker, no way to close `stake()`, and no exit at any price before maturity	ZenReferralStaking.sol	MINOR	Access control / liveness (created by the renouncement)	ACKNOWLEDGED
ZS-12	No rescue path of any kind: mis-sent tokens, the residual reward budget, and a position on a key sent to an address that cannot call back are all unrecoverable	ZenReferralStaking.sol	MINOR	Fund safety / liveness	ACKNOWLEDGED
ZS-13	A swallowed `claimFor` leaves a stale, lower rate snapshot on every open position, with no event and no way for the user to know	ZenReferralStaking.sol	MINOR	Logical issue / cross-contract integration	ACKNOWLEDGED
ZS-14	The position is a bearer instrument attached to the key: an ERC-721 approval confers authority over the staked $ZEN, and a key sale is a race for the accrued reward	ZenReferralStaking.sol	MINOR	Value binding / MEV	ACKNOWLEDGED
ZS-15	Accrual has no maturity cap: a matured position keeps earning at its snapshotted rate indefinitely	ZenReferralStaking.sol	MINOR	Arithmetic / economics	ACKNOWLEDGED
ZS-16	`withdraw` pays the final accrued reward without calling `_payReferral` and without emitting `RewardClaimed`	ZenReferralStaking.sol	INFORMATIONAL	Accounting / observability	ACKNOWLEDGED
ZS-17	`withdraw` never checks that the position is open, and zeroing the slot discards its tier — a repeat call is stopped only by the token's zero-transfer guard	ZenReferralStaking.sol	INFORMATIONAL	Insufficient validation	ACKNOWLEDGED

ID	TITLE	CONTRACT	SEVERITY	CATEGORY	STATUS
ZS-18	The constructor validates less than the setters it feeds, and `minStake`, `maxStake` and `lockPeriods` never had setters at all	ZenReferralStaking.sol	INFORMATIONAL	Insufficient validation	ACKNOWLEDGED
ZS-19	With `depositTokens` permanently uncallable, every future top-up is a bare ERC-20 transfer that emits no `TokensDeposited` event	ZenReferralStaking.sol	INFORMATIONAL	Observability (created by the renouncement)	ACKNOWLEDGED
ZS-20	A referral line terminating at the treasury is dropped rather than compressed, and nothing is emitted for an unpaid line	ZenReferralStaking.sol	INFORMATIONAL	Logical issue / observability	ACKNOWLEDGED
ZS-21	`stake` credits the requested amount with no received-amount check — verified harmless for $ZEN, and now permanently pinned	ZenReferralStaking.sol	INFORMATIONAL	Token integration	ACKNOWLEDGED
ZS-22	Verification notes: the accrual arithmetic is correct; the referral loop round-trips the sponsor chain through addresses; the early-withdrawal branch charged an undocumented second penalty and is now dead code	ZenReferralStaking.sol	INFORMATIONAL	Code quality	ACKNOWLEDGED

System Overview

`ZenReferralStaking` is a self-contained staking contract with exactly two external dependencies: the `$ZEN` ERC-20 it custodies, and the `ZenFiNftKey` marketing core it consults for ownership, level and the sponsor chain. Both are audited separately.

ZenReferralStaking.sol — NFT-key-bound tiered staking

Positions are stored under `nftId` and the actor is resolved live via `core.ownerOf(nftId)`, so a key transfer carries the whole position to the new owner; only the current owner can stake, claim or withdraw, and staking requires the key to be at level 5 or above. A stake **snapshots** its annual rate — the tier's base rate plus, on the 1-year tier only, +2 percentage points per NFT level above level 5 — and **every** claim re-snapshots it, so a level-up takes effect on the next claim with no retroactive recompute and no unbounded historical accounting. On top of the claim, 5% of the reward is paid from the pool to each of up to three sponsor lines. Four lock tiers are configured (30 / 90 / 180 / 365 days at 30 / 40 / 50 / 100 %); rewards are claimable at any time, principal only at maturity.

The implementation is disciplined: checks-effects-interactions ordering is correct on all four value-moving paths, each carries `nonReentrant`, `SafeERC20` is used throughout, `core` is `immutable`, and the accrual arithmetic was verified to produce exactly the advertised percentage with no reachable overflow. The findings below are therefore concentrated not in the code's mechanics but in its **economics and its permanence**: one unreserved balance backing both principal and rewards, a referral leg that is not fail-soft, and a configuration that — by design — can never be changed again.

Ownership renouncement — verified on-chain

This audit did not take the renouncement on trust. Every `RoleGranted` and `RoleRevoked` event the contract has ever emitted was replayed from the deployment block to block 113,225,000 — **nine events, the complete history**. The sequence is: the constructor granted `DEFAULT_ADMIN_ROLE` and `ADMIN_ROLE` to the deployer and `CORE_ROLE` to the `ZenFiNftKey` core; both deployer roles were handed to the admin account and revoked minutes later; and on 2026-07-31 the admin account revoked `ADMIN_ROLE` (block 113,221,088) and then `DEFAULT_ADMIN_ROLE` (block 113,221,152) from itself.

Result: `ADMIN_ROLE` and `DEFAULT_ADMIN_ROLE` have no holder. `DEFAULT_ADMIN_ROLE` is the role-admin of every role in this contract and no other grant path exists, so no role can ever be held again. `CORE_ROLE` remains with the core contract alone and unlocks only `claimFor`, which pays a user their own accrued rewards. Every finding in this report that concerns admin power is assessed against that fact: three are **closed permanently** by it (ZS-06, ZS-07, ZS-08), and the report is equally explicit about what the renouncement costs — no pause, no parameter correction, no early-exit valve and no rescue path, ever (ZS-11, ZS-12).

Dependencies — $ZEN and ZenFiNftKey

`$ZEN` (`TaxedToken`) is the staked and reward asset. Its transfer tax applies **only** to DEX buys and sells; wallet-to-wallet transfers are untaxed, and the staking contract is fee-excluded in addition — so the amount a staker sends is exactly the amount credited, and the contract's lack of a received-amount check (ZS-21) is provably harmless for this token. With `setStakingToken` permanently uncallable, that guarantee is now pinned for the life of the contract.

`ZenFiNftKey` supplies four live inputs with no local caching and no independent validation: `ownerOf` (who may act), `levelOf` (eligibility and the APR boost), `getUplineChain` (who receives the referral share) and `treasury` (which line is skipped). The core's own administration is **not** renounced — verified on-chain — so the staking contract's trust-minimisation is bounded by the core's (ZS-05). The core calls back into staking exactly once, through `claimFor` on a level change, and wraps that call in `try/catch` so a staking failure can never block a key purchase or upgrade — a sound isolation choice whose one side effect is that such a failure is silent (ZS-13).

Privileged Roles & Centralization

The contract uses OpenZeppelin `AccessControl` with two roles beyond `DEFAULT_ADMIN_ROLE`. The table records what each could do and its state as verified on-chain against the contract's complete role history; the powers listed for `ADMIN_ROLE` are recorded for completeness and are **permanently unreachable**.

ROLE	CONTRACT	POWERS
DEFAULT_ADMIN_ROLE	ZenReferralStaking.sol	**No holder — permanently.** Revoked from the deployer at deployment and from the admin account in tx `0xa9a3338c…` (block 113,221,152). It is the role-admin of every role in this contract and there is no other grant path, so **no role can ever be granted again.**
ADMIN_ROLE	ZenReferralStaking.sol	**No holder — permanently** (revoked in tx `0xe2255de1…`, block 113,221,088). Would have gated all nine privileged functions: `withdrawTokens` (unreserved transfer of the whole balance), `setStakingToken`, `setBaseRewardRates`, `setReferralPercent`, `setEarlyWithdrawalPenalties`, `toggleEarlyWithdrawal`, `depositTokens`, `pause`, `unpause`. All are now uncallable by anyone.
CORE_ROLE	ZenReferralStaking.sol	Held by exactly one address: the `ZenFiNftKey` core `0x74af0f87…`, granted in the constructor and now irrevocable. It unlocks **only** `claimFor(user)`, which realises that user's own accrued rewards to that user and re-snapshots their rate on a level change. It cannot move principal, change any parameter, or pay anyone other than the position's current owner and their sponsors.

Detailed Findings

ZS-01

Staked principal and the reward budget share one unreserved balance: no solvency invariant, no global stake cap, and — after the renouncement — no lever left to throttle liabilities

MAJOR

SEVERITY	MAJOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Reward-pool solvency
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:78 (`nftTotalStaked`, the only accounting state), 195-212 (`stake`, per-key cap at 199), 216-229 (`claimReward`, transfer at 225), 252-270 (`_claimAll`, transfer at 267), 275-286 (`_payReferral`, transfer at 283), 290-322 (`withdraw`, transfer at 319), 162-166 (`_accrued`), 151-158 (`effectiveRate`); frozen levers at 345-349, 352-356, 376-382

DESCRIPTION

The contract holds exactly one pot — `stakingToken.balanceOf(address(this))` — and nothing in the file distinguishes staked principal from the reward budget. Line 209 moves principal in; lines 225, 267, 283 and 319 move rewards, referral bonuses and principal out. The only accounting state is `nftTotalStaked` (line 78), which is per-key and is read in exactly one place, the `maxStake` guard at line 199; there is no global `totalStaked`, no reward-budget accumulator, and no check of the form `balance - Σprincipal ≥ payout` anywhere before a payout.

Liability accrues unconditionally. `_accrued` (162-166) is a pure linear function of elapsed time at the snapshotted rate, with no ceiling and no dependence on the remaining balance. At the live parameters the 1-year tier pays 100% base plus up to +14 percentage points of level boost, and `_payReferral` adds up to 3 × 5% of every claim as pure additional outflow — so a top-tier position costs the pool up to **131% of its principal per year**. Incoming liability is capped only per key (`maxStake` = 1,000,000 ZEN at line 199); there is no aggregate cap, so nothing on-chain relates how much may be staked to how much reward budget exists.

Before the renouncement four levers could have contained this: `pause` (376) to stop new inflows, `setBaseRewardRates` (345) to cut the APR, `setReferralPercent` (352) to cut the surcharge, and `depositTokens` (326) to refill. All four are now permanently uncallable. **The one mitigation that survives is permissionless and complete**: because no payout path consults an internal ledger, an ordinary ERC-20 transfer of `$ZEN` to the contract increases payable capacity exactly as `depositTokens` did — so anyone can fund the pool at any time (see ZS-19).

SCENARIO / IMPACT

Measured on-chain at block ~113,225,000: the contract holds **1,215,921 ZEN**, of which **223,387 ZEN** is open principal across 58 keys (summed from `nftTotalStaked` over all 1,234 minted keys), leaving a free reward buffer of **992,534 ZEN**. Reading the entire current book at the worst case — every position at the 1-year tier on a level-12 key — annual outflow would be 223,387 × 1.311 = **292,861 ZEN**, so the buffer covers roughly **3.4 years**. The pool is comfortably solvent today.

The risk is prospective and unbounded. The same buffer funds only about **757,000 ZEN** of top-tier principal for one year — less than the 1,000,000 ZEN a **single key** is permitted to stake at line 199. A book that grows past that point begins paying rewards out of later stakers' principal, and because rewards are claimable from the first second while principal is locked for up to 365 days (ZS-11), the shortfall surfaces as failed claims and blocked withdrawals in first-come-first-served order. No on-chain party could intervene: `pause` , `setBaseRewardRates` and `setReferralPercent` all revert for every caller.

RECOMMENDATION

No code change is possible; the controls are off-chain and are the load-bearing part of this product.

(1) Publish and continuously monitor the solvency figure the contract cannot express itself: `balanceOf(staking) − Σ open principal` , reconstructed from `StakeCreated` / `StakeWithdrawn` events, together with the implied runway in days at the current weighted APR. **(2)** Pre-fund each accepted position for its **full term at 115% of nominal**, by plain `$ZEN` transfer, rather than topping up after a shortfall is observed — the referral surcharge is unavoidable and, per ZS-04, effectively always payable. **(3)** Gate marketing and the front end on the funded capacity, treating it as a hard business constraint rather than a soft target. **(4)** State publicly that top-ups are permissionless and irreversible, and that there is no administrator who can rebalance the pool. **(5)** For any successor contract: track total principal globally, refuse a `stake` whose full-term reward cannot be covered by the free budget, make the reward leg best-effort instead of atomic, and keep the emergency controls behind a timelock rather than renouncing them.

RESOLUTION — ACKNOWLEDGED

Recorded as a **funding obligation, not a code defect**, and acknowledged as inherent to a pre-funded fixed-APR pool. It is deliberately reported at **Major** because it is the one path by which a user could lose principal without any attacker or privileged failure, and because the renouncement removed every on-chain response. The mitigating facts are equally deliberate: the pool is over-collateralised against the current book by roughly 3.4 years, and the funding remedy is permissionless and survives the renouncement intact.

ZS-02

`withdraw`

pays principal and reward in one all-or-nothing transfer, and there is no principal-only or partial exit

MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Logical issue / fund safety
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:290-322 (`withdraw`), specifically 299-300 (`originalAmount` , `rewardAmount`), 308 (`returnedPrincipal`), 319 (the single `safeTransfer`); contrast 216-229 (`claimReward`)

DESCRIPTION

`withdraw` computes `returnedPrincipal` (308) and `rewardAmount` (300) and sends them as a **single** `safeTransfer` at line 319. There is no amount parameter, no flag to forgo the reward, and no emergency path anywhere in the contract that returns principal alone. If the balance is short by one wei of `returnedPrincipal + rewardAmount` , the transfer reverts and the staker receives nothing — including the portion of their own principal that is demonstrably still held by the contract.

This is a robustness gap rather than a loss in itself: it only bites in the shortfall state described in ZS-01, and there is a workaround. A staker can call `claimReward` first, which resets `lastClaimTimestamp` (222) and shrinks the accrued reward to approximately zero, after which `withdraw` needs only the principal. That workaround is itself gated slightly harder than it looks, because a claim must also fund the referral surcharge (ZS-03) — it needs up to 1.15 × the reward, whereas the direct withdrawal needs principal + reward. In every realistic state the claim-then-withdraw route is the cheaper one for the staker, and it should be documented as the recommended exit when the pool is thin.

The second half of the picture is that there is **no early exit at all.** `isEarlyWithdrawalEnabled` is false and `toggleEarlyWithdrawal` (371) can never be called again, so line 297 rejects every withdrawal inside the lock window unconditionally — a staker who watches the pool drain cannot exit at any price, even by accepting the configured 75% tier-3 penalty, until the term matures (see ZS-11).

SCENARIO / IMPACT

A 100,000 ZEN position on the 1-year tier at level 12 accrues at 114% APR. Suppose the free buffer has fallen below the position's full claim. At maturity `withdraw` needs 100,000 + 114,000 = 214,000 ZEN in one transfer and reverts if the contract holds less — even while holding, say, 150,000 ZEN, more than enough to return the principal. The staker's correct move is `claimReward` first: it needs 114,000 × 1.15 = 131,100 ZEN and, once it lands, `withdraw` needs only the 100,000 principal. Nothing in the contract or its errors tells the user this; the revert surfaces as a bare `SafeERC20` failure.

RECOMMENDATION

No code change is possible. Document the exit ordering explicitly in the staking UI and in the user-facing terms: **claim first, then withdraw**, because a claim requires roughly 1.15 × the accrued reward while a direct withdrawal requires principal + reward in one atomic transfer. Surface the contract's live balance and the position's `calculateReward` next to the withdraw button so a user can see whether their exit will fit. For any successor: split `withdrawPrincipal` from the reward claim so principal can never be blocked by unpaid rewards, and make the reward leg partial (`min(accrued, availableBudget)`) rather than all-or-nothing.

RESOLUTION — ACKNOWLEDGED

Acknowledged. The behaviour is a direct consequence of ZS-01 and is only reachable in the shortfall state described there; it is reported separately because it changes the **recommended user action** and because the absence of a principal-only exit is what turns a temporary shortfall into a blocked withdrawal. The renouncement neither created nor removed the mechanism, but it removed every operator remedy — there is no longer anyone who can rebalance the pool or pause reward outflow while principal is short.

ZS-03

The referral leg is not fail-soft: a claim reverts unless the pool holds 115% of the reward, and the core's `try/catch` turns that into a silent no-op MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Reward-pool solvency / denial of service
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:275-286 (`_payReferral` , unguarded `safeTransfer` at 283), called from 226 (`claimReward`) and 268 (`_claimAll`); rate re-snapshot bundled into the same transaction at 223 and 260; cross-contract: `ZenFiNftKey.sol` :694 and 989 (`try referralStaking.claimFor(…) {} catch {}`)

DESCRIPTION

`_payReferral` transfers `each = reward * referralPercentBps / BPS` to every real sponsor with a bare `safeTransfer` at line 283 — inside a loop, with no balance check, no clamp and no `try/catch` . The referral leg runs **after** the staker has already been paid (225 then 226; 267 then 268), so when the pool is short the staker's transfer succeeds and a sponsor transfer reverts, unwinding the entire transaction atomically.

The practical consequence is that a claim requires `reward × (1 + 0.05n)` in the pool, where `n` is the number of non-zero, non-treasury sponsors — up to **1.15 ×** the reward. A staker whose reward is fully covered by the balance still cannot claim a single wei of it. This is a deliberate divergence from the rest of the protocol: `ZenFiNftKey` 's buyer- `$ZEN` leg explicitly clamps to the available balance and never reverts, and the core wraps the staking call itself in `try/catch` .

The second half of the defect is that the rate re-snapshot at line 260 is welded into the same transaction. The core invokes `claimFor` on a level change and swallows every error (`ZenFiNftKey.sol` :694, and 989 on the promo path), so a failed claim leg is invisible: the key levels up on-chain, the user pays for the upgrade, and the staking position silently keeps its **old, lower** rate with no event and no revert to alert anyone. The accrual itself is preserved and any later successful claim restores the correct rate, so the loss is bounded by the rate delta over the interval — but the user has no way to know it is happening (see ZS-13).

SCENARIO / IMPACT

The free buffer is 40,000 ZEN. Bob's position has accrued 38,000 ZEN and he has three real sponsors, so the claim needs 38,000 + 3 × 1,900 = 43,700 ZEN. `claimReward` transfers 38,000 to Bob at line 225, pays two sponsors at line 283, then reverts on the third — the whole call unwinds and Bob receives nothing, even though 38,000 ZEN was available and unambiguously his. He can still realise a **smaller** position of his own (`claimReward` is per-index), but not this one.

Bob then buys a level-6 upgrade. The core sets his level, calls `claimFor` , and the same shortfall reverts it inside the `catch {}` . His purchase succeeds, his key is level 6, and his 1-year-tier position keeps the level-5 rate until he manages a successful claim. Nothing on-chain records that the auto-claim was skipped.

RECOMMENDATION

No code change is possible. Keep the pool funded to at least **1.15 × the aggregate unclaimed accrual**, tracked off-chain from `StakeCreated` and `RewardClaimed`, so the referral leg always fits. Monitor for the silent-skip condition by comparing each position's stored `rate` (via `getStakes`) against `effectiveRate(nftId, tier)` after every level change on the core, and prompt affected users to claim. For any successor: clamp each sponsor payment to the remaining balance and tolerate a failed sponsor transfer rather than reverting the staker's own claim, and decouple the rate re-snapshot from the payout so a level-up always updates the rate even when the transfer leg cannot run.

RESOLUTION — ACKNOWLEDGED

Acknowledged. The precondition is the shortfall state of ZS-01 and the two must not be severity-summed. `setReferralPercent(0)` (352) was the single parameter that could have removed the 15% overhead and unblocked claims during a shortfall; it now has no possible caller, so the 500 bps setting is fixed for the life of the contract.

ZS-04

`_payReferral`

applies no sponsor qualification, so a staker can capture the whole 15% surcharge with a three-key upline of their own

MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Logical issue / economics
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:272-286 (`_payReferral`), 278 (`core.getUplineChain(owner, 3)`), 282 (only `address(0)` and the treasury are skipped); cross-contract: `ZenFiNftKey.sol`:1030-1049 (`getUplineChain` walks the raw `sponsorOf` genealogy), 599 and 965 (`sponsorOf` fixed at mint), 786-798 (the core's own affiliate payout uses the level-qualified `parentAt` chain instead)

DESCRIPTION

`_payReferral` pays 5% of the claim to every entry of `core.getUplineChain(owner, 3)` and applies exactly two filters at line 282: the zero address and the treasury. It never checks the sponsor's level, never checks whether the sponsor has staked, and never checks whether the sponsor is active in any sense.

That is materially looser than the marketing core's own affiliate rules, which pay along the **level-qualified** `parentAt` placement chain — a chain the core explicitly compresses past any ancestor below the purchased level. The staking leg instead walks the raw registration genealogy `sponsorOf`, where a dormant level-1 key qualifies for 5% of every downline staking claim indefinitely.

Because the surcharge is paid from the pool rather than deducted from the claimer (lines 65-67 document this deliberately: *"Paid on top of the claim, from the pool — the claimer keeps 100%"*), a self-owned upline costs its operator nothing in foregone yield — it is pure additional pool subsidy. Registering three of one's own wallets above one's staking wallet costs three level-1 keys, and the resulting chain is permanent because `sponsorOf` is written once at mint. The incremental harm over ZS-01 is bounded at 15% of the operator's own legitimately accrued reward — this is not theft and a non-staker can extract nothing — but it means the pool must be budgeted as though **every** staker captures the full 1.15 × factor, because in practice any staker can.

SCENARIO / IMPACT

With `basePriceUsd` verified on-chain at $10 and per-level doubling, an operator registers three wallets holding level-1 keys chained one under the other ($10 each), then registers their staking wallet under the third and upgrades it to level 5 — which is required in any case by the `MIN_STAKE_LEVEL` gate at line 197. The marginal cost of the sybil upline is **$30**.

The staking wallet stakes 100,000 ZEN on the 1-year tier. After one year `claimReward` pays it 100,000 ZEN at line 225, then `_payReferral` pays 5,000 ZEN to each of the three upline keys at line 283 — all controlled by the same operator. The operator's wallets receive 115,000 ZEN in total, and the pool has funded 115% of the position's nominal APR. Nothing in the contract distinguishes this from an organic three-line downline.

RECOMMENDATION

No code change is possible. **Budget the reward pool at 115% of nominal APR for every position**, not at 100% — that is the only assumption that is robust, and it is the single most actionable consequence of this finding for ZS-01. Monitor for the pattern off-chain (a claim whose three `ReferralPaid` recipients are all shallow, never-staking keys registered in a tight block window) so it can at least be measured. For any successor: gate each sponsor on real participation — require `levelOf(sponsorKey) >= MIN_STAKE_LEVEL`, or a non-zero staked balance, or derive the chain from the core's level-qualified `parentAt` placement chain the way the core's own affiliate payout does.

RESOLUTION — ACKNOWLEDGED

Acknowledged. No sponsor-qualification parameter ever existed, so the renouncement neutralised nothing here; the one lever that could have shrunk the exposure — `setReferralPercent` (352) — is now permanently uncallable, so the 15% self-payable surcharge is frozen into the contract for its entire lifetime.

ZS-05

The renouncement does not extend to `ZenFiNftKey`

: eligibility, APR, the payee and the sponsor chain are all live reads from a core whose administration is still privileged

MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Access control / cross-contract trust
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:56 (`core` is `immutable`), 116 (the sole `CORE_ROLE` grant), 131-134 (`_ownerOrRevert` → `core.ownerOf`), 143 (`core.levelOf` in `levelBoost`), 197 (the `MIN_STAKE_LEVEL` gate), 244 and 253 (`core.nftIdOf` / `core.ownerOf`), 278-279 (`core.getUplineChain` , `core.treasury`); cross-contract: `ZenFiNftKey.sol` :487-490 (`setReferralStaking`), 932-994 (`redeemPromo` , `levelOf` written at 977)

DESCRIPTION

`ZenReferralStaking` performs **no** independent validation of any economic input. Eligibility to stake (line 197), the APR boost (143 → 145 → 155), the payee of every reward (132, 253) and the recipients of the referral share (278-279) are all live reads from `ZenFiNftKey`. The `core` address is `immutable`, which correctly forecloses repointing it — but `CORE_ROLE` was granted to the core in the constructor and, with `DEFAULT_ADMIN_ROLE` now unheld, can never be revoked either.

The consequence is that renouncing this contract bounds its trust-minimisation by the core's, and **the core's administration is not renounced.** This audit verified the core's role state directly on-chain: at the time of writing `ZenFiNftKey`'s `DEFAULT_ADMIN_ROLE` and `ADMIN_ROLE` are held by the account `0x1898d47b…` (a plain EOA), `ADMIN_ROLE` is additionally held by `0xF37A2CB9…`, and `PROMO_SIGNER_ROLE` by `0xC83F200C…`.

Two concrete couplings follow. **(a)** `ZenFiNftKey.setReferralStaking` (487) is `ADMIN_ROLE`-gated on the core; setting it to the zero address permanently stops `claimFor` from ever being invoked. No funds move and users can self-heal — any manual claim re-snapshots the rate — but positions would silently keep a stale, lower rate after a level-up until their owner happens to claim, and the staking contract can neither detect nor restore the hook. **(b)** The `PROMO_SIGNER_ROLE` path (`redeemPromo`, 932-994) writes `levelOf` for free, so a promo-granted key can satisfy the `MIN_STAKE_LEVEL` gate and carry the full +14-point boost without a paid purchase — the staking contract's liabilities remain influenceable by the core's administration in a way this contract can no longer answer, because `setBaseRewardRates` and `pause` are gone.

Note what is **not** at risk: `CORE_ROLE` here unlocks only `claimFor`, which pays a user their own accrued rewards to that user. It cannot move principal, cannot pay a third party, and cannot change any parameter.

SCENARIO / IMPACT

(a) A staker holds a 1-year-tier position snapshotted at rate 100 while their key is at level 5. They upgrade to level 12. If `setReferralStaking(address(0))` has been called on the core, `_applyLevel` skips the hook and the position keeps rate 100 instead of being re-snapshotted to 114. On a 100,000 ZEN position left alone for six months that is a 7,000 ZEN shortfall to the staker — recoverable at any time by calling `claimReward` themselves, but invisible until they do.

(b) A free promo raises a key to level 12. That key now passes the `MIN_STAKE_LEVEL` gate at line 197 and snapshots `effectiveRate = 114` on the 1-year tier, drawing on the same pool as every paid key. `ZenReferralStaking` has no lever left: the rate cannot be cut, new stakes cannot be stopped, and the contract cannot be paused.

RECOMMENDATION

Extend the renouncement disclosure to the dependency. Publish, alongside this contract's renouncement, the **current holders of `ZenFiNftKey`'s `DEFAULT_ADMIN_ROLE`, `ADMIN_ROLE` and `PROMO_SIGNER_ROLE`**, their custody model, and whether they sit behind a multisig or timelock — because this contract's economics are now strictly downstream of them, and a reader who sees only the staking renouncement will overestimate the guarantee. If those core powers are not in active use, place them behind a timelock or renounce them too. Monitor `PromoRedeemed` on the core for high-`toLevel` redemptions followed by large `StakeCreated` events, which is the only remaining detection surface. For a successor: snapshot the level from a source the staking contract controls, or bound the boost independently of the core.

RESOLUTION — ACKNOWLEDGED

Acknowledged as an inherent property of a contract that delegates its economic inputs to a dependency. The renouncement did not touch these levers and removed the two counter-measures that existed: `CORE_ROLE` can no longer be revoked, and `setBaseRewardRates` can no longer compensate. Reported here so that "ownership renounced" is read with the correct scope — it is a complete statement about `ZenReferralStaking` and not about the system it depends on.

ZS-06

`withdrawTokens`

allowed the admin to transfer the entire balance — staked principal included — with no reserve check and no event

MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Access control / centralization
STATUS	FIXED
LOCATION	contracts/staking/ZenReferralStaking.sol:331-334 (`withdrawTokens`), the sole balance check at 332, no event; affected payout paths at 225, 267, 283 and 319

DESCRIPTION

`withdrawTokens` performed exactly one check — `amount <= stakingToken.balanceOf(address(this))` at line 332 — and then transferred to the caller. It maintained no reserve for outstanding stakes, which is unsurprising given that the contract keeps no global principal accumulator at all (ZS-01): the only accounting state, `nftTotalStaked`, is per-key and is never consulted here. The check therefore treated **staked user principal as admin-withdrawable**, and the double-count was written into the code rather than merely implied by it.

It also emitted **no event.** `depositTokens` (326-329) emits `TokensDeposited`, but its counterpart is silent, so a full drain would have left no protocol-level trace beyond the raw ERC-20 `Transfer`. Together with ZS-07 this was the contract's principal centralization risk: a single privileged transaction could have removed every staker's principal, after which every `withdraw`, `claimReward` and `_claimAll` would revert on an insufficient balance and no staker would recover anything.

SCENARIO / IMPACT

Counterfactual, at the live balance: the `ADMIN_ROLE` holder calls `withdrawTokens(1_215_921e18)`. Line 332's only check passes because the amount equals the balance, and line 333 transfers the entire pool — 992,534 ZEN of reward budget **and** 223,387 ZEN of user principal — to the caller. Every subsequent payout reverts inside the token on an insufficient balance. There is no event to alert a monitor, and no reserve accounting that could have refused the call.

RECOMMENDATION

For any successor: track total staked principal globally and require `balance - amount >= totalPrincipal` in any administrative withdrawal, emit an event, and place the function behind a timelock. On this deployment no action is possible or required — the capability no longer exists.

RESOLUTION — FIXED

Permanently neutralised by the on-chain renouncement. `withdrawTokens` is `onlyRole(ADMIN_ROLE)`. The contract's complete nine-event role history shows `ADMIN_ROLE` revoked from the deployer at deployment and from the admin account in tx `0xe2255de1…` (block 113,221,088), leaving **no holder**; `DEFAULT_ADMIN_ROLE` — the role-admin of every role, and the only path by which `ADMIN_ROLE` could ever be granted again — was revoked in tx `0xa9a3338c…` (block 113,221,152). The function is uncallable by anyone, forever, and **no administrative drain of this contract is possible**. This is the single largest risk the renouncement removed.

ZS-07

`setStakingToken` could re-denominate every open position while the principal stayed in the old token

MEDIUM

SEVERITY	MEDIUM
CONTRACT	ZenReferralStaking.sol
CATEGORY	Access control / centralization
STATUS	FIXED
LOCATION	contracts/staking/ZenReferralStaking.sol:336-342 (`setStakingToken`), consumed by every transfer at 209, 225, 267, 283 and 319; the test that covers only the benign case at test/ZenReferralStaking.ts:331-341

DESCRIPTION

`setStakingToken` validated only that the new address was non-zero and different from the current one (337-339). It did **not** require that nothing was staked, did not migrate balances, and did not touch `nftTotalStaked` or any `Stake` record. Because every transfer in the contract dereferences the mutable `stakingToken` storage slot at call time, flipping it would have made every subsequent payout denominate in the new token while the entire staked principal remained stranded in the old one — and every subsequent `stake` would have pulled the new token instead.

The project's own test exercises the repoint only *"while nothing is staked"*, a condition the contract never enforced. The quiet variant of the abuse is worse than the loud one: stakers calling `withdraw` would have received their full **nominal** principal denominated in a worthless token, while the real `$ZEN` sat untouched and recoverable through ZS-06.

A secondary consequence disappears with it. `stake` credits the caller-supplied `_amount` without measuring what actually arrived (ZS-21); that is provably harmless for `$ZEN` , but a repoint to a fee-on-transfer or rebasing token would have turned the unmeasured credit into systematic over-crediting paid out of other stakers' principal.

SCENARIO / IMPACT

Counterfactual: with positions open, the `ADMIN_ROLE` holder calls `setStakingToken(worthlessToken)` at line 340. From that block, `withdraw` at line 319 transfers `returnedPrincipal + rewardAmount` of the worthless token to each exiting staker, who sees the correct **number** arrive; the `$ZEN` that backed those positions never moves and remains available to `withdrawTokens` . No event on this contract distinguishes the change from a legitimate migration except `StakingTokenUpdated` , which does not signal that positions are open.

RECOMMENDATION

For any successor: forbid changing the staking token while any principal is staked, or version positions by token so an old position always settles in the asset it was created in; place the setter behind a timelock. On this deployment no action is possible or required.

RESOLUTION — FIXED

Permanently neutralised by the on-chain renouncement. `setStakingToken` is `onlyRole(ADMIN_ROLE)` , no account holds that role, and none can ever be granted (see ZS-06 for the verified role history). `stakingToken` is pinned to `$ZEN` (`0xCF361DE2…`) for the life of the contract, which also permanently pins the token-integration guarantee described in ZS-21.

ZS-08

Unbounded rate, referral and penalty setters could have rewritten the economics of open positions — including a rate that bricks a position through checked-arithmetic overflow

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Access control / insufficient validation
STATUS	FIXED
LOCATION	contracts/staking/ZenReferralStaking.sol:345-349 (`setBaseRewardRates` , length check only), 352-356 (`setReferralPercent` , `bps <= BPS`), 358-369 (`setEarlyWithdrawalPenalties` , `<= BPS` at 365), 371-374 (`toggleEarlyWithdrawal`); interacting with 162-166 (`_accrued`) and the rate snapshots at 202, 223 and 260

DESCRIPTION

Four validation gaps existed on the administrative surface, recorded together because each defines part of what the renouncement bought.

(a) `setBaseRewardRates` validated only the array length, with no upper bound on any rate. Since `_accrued` computes `s.amount * elapsedSeconds * (s.rate * 100)` in checked arithmetic (164-165), a sufficiently large rate makes that product overflow and revert with `Panic(0x11)` . Because `_accrued` is called by both `claimReward` (219) **and** `withdraw` (300), any position that had re-snapshotted the poisoned rate would have been permanently unrecoverable — and lowering the rate again would not have helped, because the only paths that rewrite `s.rate` (223, 260) must themselves complete a claim first. At the live values there is no such risk whatsoever: the numerator peaks near 3.6×10^{35} over a one-year horizon, dozens of orders of magnitude below the `uint256` ceiling.

(b) `setReferralPercent` bounded only at `bps <= BPS` , i.e. up to 100% **per line**, so up to 300% of every claim could have been added as surcharge. **(c)** `setEarlyWithdrawalPenalties` capped each penalty at exactly `BPS` ; at that value the principal penalty consumes the whole position and the payout becomes a zero-value transfer, which `$ZEN` rejects outright — so an early withdrawal in that tier would always have reverted. The correct cap is `< BPS` . **(d)** `toggleEarlyWithdrawal` is a blind toggle rather than an explicit setter, so two racing administrative transactions could leave the flag in the unintended state.

SCENARIO / IMPACT

Counterfactual, for the most severe gap (a): with `ADMIN_ROLE` still held, the admin calls `setBaseRewardRates([30, 40, 50, 1e50])` . A user then stakes or claims on tier 3 and lines 202/223/260 snapshot the poisoned rate into their position. `_accrued` then evaluates `amount * elapsed * (1e50 * 100)` , which for a 1,000,000 ZEN position overflows within roughly twelve seconds of accrual. From that moment `claimReward` reverts at line 219 and `withdraw` reverts at line 300, and the position is frozen with no recovery path.

RECOMMENDATION

For any successor: bound `setBaseRewardRates` at a sane ceiling, bound `setReferralPercent` so `REFERRAL_LEVELS × bps` cannot exceed a small fraction of the claim (the live 500 bps × 3 lines = 15% is the sensible figure), tighten the penalty cap to `< BPS` , replace the toggle with `setEarlyWithdrawal(bool)` , and place all of them behind a timelock. On this deployment no action is possible or required.

RESOLUTION — FIXED

Permanently neutralised by the on-chain renouncement. All four functions are `onlyRole(ADMIN_ROLE)` ; no account holds that role and none can be granted (see ZS-06). The rates `[30, 40, 50, 100]` , the referral share of 500 bps, the penalties `[3000, 4500, 6000, 7500]` and `isEarlyWithdrawalEnabled = false` are now immutable constants of the system, so no position's economics can ever be rewritten out from under its owner and no rate can ever be poisoned. The flip side — that the same values can never be **corrected** either — is recorded in ZS-10 and ZS-11.

ZS-09

The per-key `stakes` array is append-only and never compacted, so `claimAllRewards` and the core's level-up hook grow without bound MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Denial of service / gas
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:80 (`stakes`), 205 (unconditional `push`), 310-314 (`withdraw` zeroes the fields but never `pop` s), 252-270 (`_claimAll` , full-length loop at 255 with a live `core` call per iteration), 231-235 (`claimAllRewards`), 243-247 (`claimFor`); cross-contract: `ZenFiNftKey.sol` :624-627 (`purchase` loops `_applyLevel` once per level) and 693-695 / 988-990 (the `try/catch` hook)

DESCRIPTION

`stake` caps only the **summed** principal per key (`maxStake` at line 199) and never the **number** of positions; with `minStake` at 1 ZEN a key can hold arbitrarily many. Line 205 always pushes a new slot and never reuses one, and `withdraw` zeroes the five fields (310-314) without popping, so `stakes[nftId].length` only ever grows — a withdrawn slot still costs a cold storage read on every subsequent `_claimAll` iteration.

`_claimAll` iterates the full length on every call, and it is what the core invokes through `claimFor` on every level-up — once **per level**, because `purchase` loops `_applyLevel` per level crossed. The core's `try/catch` does not fully insulate the parent transaction: under the 63/64 gas rule the child consumes almost all remaining gas before the `catch` runs, so on a multi-level purchase a sufficiently bloated hook can leave too little gas for the remaining iterations and revert the whole purchase.

The reachability is limited and the finding should be read accordingly. `stake` is gated by `_ownerOrRevert` (196), so **only the key's own owner can bloat their own array** — this is self-griefing, not something that can be inflicted on a third party. Nor does it revert "at any gas limit": the parent fails only when the supplied gas limit is below the bloated requirement, and gas estimation adapts. The real consequences are that a key's own upgrade path becomes progressively more expensive and eventually estimation-dependent, and that `claimAllRewards` for that key can be pushed past the block gas limit — after which per-index `claimReward` and `withdraw` still work, so no funds are lost.

SCENARIO / IMPACT

RECOMMENDATION

No code change is possible. The front end should read `getStakes(nftId).length` , warn above roughly 50 positions, and steer users toward consolidating into fewer, larger positions — both because upgrades get more expensive and because `claimAllRewards` eventually stops fitting in a block. Support should know that per-index `claimReward` / `withdraw` remain available for a bloated key. For any successor: cap the number of positions per key, swap-and-pop on full withdrawal so the array shrinks, and expose a paginated `claimRange(nftId, from, to)` with the core hook calling a bounded variant.

RESOLUTION — ACKNOWLEDGED

Acknowledged. There was never a position cap, a `pop` or an administrative lever addressing this, so the renouncement did not create it — but it removes the containment options: `pause` can no longer stop further position creation on a key that is being bloated, and `minStake` (which has no setter at all) can never be raised to make it more expensive.

ZS-10

`levelBoost`

contradicts its own NatSpec and 4 of the 20 shipped tests: level 5 earns +0%, not the documented +2%, and level 12 earns +14%, not +16%

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Code quality / documentation
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:145 (`(level - MIN_STAKE_LEVEL) * MULTIPLIER`) against 138-141 (NatSpec: `(level - (MIN_STAKE_LEVEL - 1)) * MULTIPLIER` , *"level 5 → +2%, level 12 → +16%"*) and 25-29 (the header block, same claim); applied at 155 via `effectiveRate` and snapshotted at 202, 223 and 260. Failing assertions at test/ZenReferralStaking.ts:75, 286, 398, 418

DESCRIPTION

Line 145 computes the boost as `(level - MIN_STAKE_LEVEL) * MULTIPLIER`. With the live constants that yields **0 at level 5**, +2 at level 6 and **+14 at level 12**. Both prose statements in the deployed source say otherwise: lines 138-141 give the formula `(level - (MIN_STAKE_LEVEL - 1)) * MULTIPLIER`, and lines 25-29 state explicitly that the boost *"starts AT MIN_STAKE_LEVEL"*, is *"exactly +MULTIPLIER% (one step)"* there, and reaches +16% at level 12. Only line 23 of the header agrees with the code, so the header block contradicts itself.

Running the project's own suite against the deployed source confirms the divergence: **16 passing, 4 failing** in `test/ZenReferralStaking.ts`, with every failure being a level-boost expectation asserting the superseded formula (the `40 vs 42` failures come from the test fixture's own base rates, not from the live `[30, 40, 50, 100]`).

The user-facing impact is nil, and that is the important qualification. The ZenFi front end computes the boost as `(level - minBoostLevel) * multiplier` — matching the **code**, not the stale comment — and reads `MULTIPLIER` and `MIN_STAKE_LEVEL` from the chain, so the APR a user is shown is the APR they receive. The published level perks (L8 +6%, L12 +14%) also match the code. The defect is therefore that the **deployed, publicly verified source misdocuments its own behaviour**, and that the shipped test suite no longer passes — which is exactly what an independent reviewer reads first.

SCENARIO / IMPACT

A prospective staker reads the verified source on BscScan, sees at lines 25-29 that *"level 5 → +2%"*, and stakes 500,000 ZEN on the 1-year tier with a level-5 key. `stake` calls `effectiveRate` at line 202: the base rate is 100 and `levelBoost` returns `(5 - 5) * 2 = 0`, so the snapshot is 100, not the documented 102. Over one year that is a 10,000 ZEN gap between the comment and the payout — and, because a claim is 15% larger than the reward, a further 1,500 ZEN the sponsors do not receive. The user was never mis-priced by the application, only by the contract's own documentation.

RECOMMENDATION

The behaviour is frozen; the documentation is not. Correct the record wherever it is still reproduced: state plainly, in the docs and in any published copy of the source, that **the boost is 0 at level 5 and adds 2 percentage points per level above it — level 6 → +2%, level 12 → +14%.** Bring `test/ZenReferralStaking.ts` back to green against the deployed behaviour (lines 75, 286, 398 and 418) so that a failing suite never again reads as a live defect. If the `+2%-at-level-5` semantics were the intended product, that is a business decision requiring a redeployment, not a comment change.

RESOLUTION — ACKNOWLEDGED

Acknowledged and permanent. `levelBoost` reads two `constant` values that live in bytecode and never had setters; the only lever that could have compensated for the missing step — raising the tier-3 base rate via `setBaseRewardRates` (345) — is permanently uncallable. Reported at **Minor** rather than higher precisely because the application layer already follows the code: no user is quoted a rate they do not receive.

ZS-11

Pausable

is permanently inert and early withdrawal is permanently disabled: no circuit breaker, no way to close `stake()` , and no exit at any price before maturity MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Access control / liveness (created by the renouncement)
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:63 (`isEarlyWithdrawalEnabled`), 296-297 (the early-withdrawal gate), 303-307 (the now-dead penalty branch), 362-369 (`setEarlyWithdrawalPenalties`), 371-374 (`toggleEarlyWithdrawal`), 376-382 (`pause` / `unpause`), and the `whenNotPaused` modifiers at 195, 216, 231 and 290

DESCRIPTION

This is the cost side of the renouncement, and it should be read together with ZS-06 through ZS-08 rather than in isolation.

`isEarlyWithdrawalEnabled` is false on-chain and `toggleEarlyWithdrawal` can never be called again, so line 297 rejects every withdrawal inside the lock window unconditionally and forever. Every staker is hard-locked for the full tier term — up to 365 days — **with no exit at any price**, not even by accepting the configured 75% tier-3 penalty. The entire penalty machinery at 303-307 and the `setEarlyWithdrawalPenalties` setter are now unreachable code, which also means the penalties reported by `getStakingInfo()` describe an exit route that does not exist.

`pause` is likewise permanently uncallable, so `whenNotPaused` on `stake` , `claimReward` , `claimAllRewards` and `withdraw` is a modifier that can never engage. If an accounting failure or a shortfall is observed, **nothing can stop new deposits entering the pool or rewards leaving it**.

The converse is equally true and worth stating plainly: the contract can never be paused **against** users either. Claims and matured withdrawals can never be frozen by anyone, and no party can prevent a staker from exiting once their term is up. A closely related consequence is that the contract's only endogenous inflow is gone: with early withdrawal permanently off, no penalty revenue can ever be collected, so the reward budget is a strictly non-increasing fund absent external top-ups (ZS-01).

SCENARIO / IMPACT

A user stakes 100,000 ZEN on the 1-year tier. On day 30 they conclude the pool is draining faster than it is being funded and would gladly pay the 75% tier-3 penalty to recover 25,000 ZEN now. Line 297 evaluates `isEarlyWithdrawalEnabled (false) || !isEarlyWithdrawal (false)` and reverts with *"Early withdrawal is not allowed"*. No party on earth can change that: `toggleEarlyWithdrawal` requires a role with zero holders and no grant path. The user must wait 335 more days and take whatever the pool can then pay. In parallel, nobody can call `pause()` to stop further deposits entering the same pool.

RECOMMENDATION

Document this prominently in the staking UI and in the terms shown **before** a user commits, since it is the single most consequential product fact: **(a)** there is no early exit under any circumstances and the penalty rates returned by `getStakingInfo()` are permanently unreachable — they should not be displayed as if they were an option; **(b)** the contract has no administrator and no pause, so users bear pool risk with no operator intervention possible, in exchange for a guarantee that no operator can touch their funds either. For any successor: make the early-exit valve permissionless and always available, with the penalty as its price, and keep emergency controls behind a timelock rather than renouncing them.

RESOLUTION — ACKNOWLEDGED

Acknowledged as a deliberate and disclosed trade-off. It is created by the renouncement and is permanent. It is the direct counterpart of ZS-06 and ZS-07: the same act that made an administrative drain impossible also removed the emergency exit and the circuit breaker. The report takes no position on whether the trade is correct — it is a governance choice — but it must be disclosed to users with the same prominence as the renouncement itself.

ZS-12

No rescue path of any kind: mis-sent tokens, the residual reward budget, and a position on a key sent to an address that cannot call back are all unrecoverable

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Fund safety / liveness
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:331-334 (`withdrawTokens`, the only outbound non-user path, now dead), 131-134 (the position follows the live NFT owner), and the absence of any `receive`, `fallback` or `rescueERC20` in lines 1-383; contrast `ZenFiNftKey.sol`:1087-1094 and 1098-1106

DESCRIPTION

The contract declares no `rescueERC20`, no `receive` and no `fallback`, and its single outbound administrative path — `withdrawTokens` — is now permanently uncallable. It is therefore a one-way sink for anything that is not a live staking position.

Three classes of value are affected. **(1)** Any ERC-20 other than `$ZEN` sent to the address is destroyed; `$ZEN` sent to it usefully becomes reward budget (ZS-19), but there is no undo. **(2)** When every position has eventually been withdrawn, whatever reward budget remains is locked in the contract forever — it can never be recovered or redeployed. **(3)** The most consequential case for users: positions are keyed by `nftId` and the actor is resolved through `core.ownerOf(nftId)`, and `ZenFiNftKey` permits contract holders. If a staker transfers their key to an address that cannot make an arbitrary call to this contract, the position becomes inaccessible — with no protocol-side recovery and no operator who could refund it off-chain.

Case (3) should be stated precisely: the position is not destroyed. It follows the key, so it is recoverable whenever the new owner can call the contract, or transfers the key onward to someone who can. The loss is unconditional only for a recipient that provably can neither call this contract nor move the token.

SCENARIO / IMPACT

A staker with a 50,000 ZEN position open on their key sends that key to a centralised-exchange deposit address, intending to sell it and unaware that the staking position rides with the NFT. `core.ownerOf` now returns the exchange's address, so the staker's own `withdraw` reverts with *"Not NFT owner"*, and the exchange's deposit contract will never call `ZenReferralStaking`. The 50,000 ZEN principal plus all accrued reward sits in the contract, recoverable only if the exchange can be persuaded to make the call or return the key. `withdrawTokens` — the only path that could once have retrieved it for a manual refund — reverts for every caller.

RECOMMENDATION

Warn at the point of transfer, not afterwards. The key-transfer flow in the UI should read `getStakes(nftId)` and hard-warn (or block) whenever any entry has `amount > 0`, stating that the staked principal moves with the key and that a recipient which cannot call this contract cannot retrieve it. Publish that **only `$ZEN` should ever be sent to the staking address**, that such a transfer is an irreversible donation to the reward pool, and that any other asset sent there is permanently lost. For any successor: include a `rescueERC20` that excludes the staking token, and consider requiring the position's owner to settle before a key with open positions can be transferred.

RESOLUTION — ACKNOWLEDGED

Acknowledged and permanent. Before the renouncement `withdrawTokens` gave an operator a custodial way to retrieve stranded value and refund an affected user off-chain; that discretion is gone — which is exactly the same property that makes an administrative drain impossible (ZS-06). Every category of stranded value in this contract is now final.

ZS-13

A swallowed `claimFor`

leaves a stale, lower rate snapshot on every open position, with no event and no way for the user to know

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Logical issue / cross-contract integration
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:243-247 (`claimFor`), 252-270 (`_claimAll` , re-snapshot at 260 bundled with the transfers at 267-268), 275-286 (`_payReferral` , transfer at 283); call sites: `ZenFiNftKey.sol` :693-695 and 988-990, both `try … {} catch {}`

DESCRIPTION

`claimFor` is the only mechanism that re-snapshots a position's rate at level-up, and both core call sites wrap it in a bare catch-all that discards the reason and emits nothing. Every revert inside it is therefore invisible on-chain.

Three failure modes are reachable: the reward transfer at line 267 reverting for lack of balance; the referral transfer at line 283 reverting because the pool covers the reward but not the additional surcharge (ZS-03); and `_claimAll` running out of gas on a bloated `stakes` array (ZS-09). Several other paths were checked and are **not** reachable — the `nftId == 0` early return at 245 is dead from both call sites because the core enforces one key per wallet, and `effectiveRate` 's tier bound at 152 cannot trigger because tiers are validated at stake time.

The consequence is bounded but silent: the key levels up, the user pays for the upgrade, and the position keeps its **old, lower** rate. The accrual itself is preserved and any later successful `claimReward` or `claimAllRewards` restores the correct rate, so the loss is the rate delta over the interval rather than the reward itself. What makes it a finding is that nothing — no event, no revert, no error — tells the user it happened. The `try/catch` isolation is itself the right design choice: a staking failure must never block a key purchase.

SCENARIO / IMPACT

A staker holds a 100,000 ZEN 1-year-tier position snapshotted at rate 100 while at level 5. With the pool thin, they buy the level 5 → 8 upgrade. The core sets `levelOf = 8` and calls `claimFor` ; `_claimAll` computes the accrued reward, attempts the transfer at line 267, reverts for lack of balance, and line 694's `catch {}` swallows it. The purchase succeeds and the key is level 8, but the position continues to accrue at 100% rather than the 106% the new level earns, and will do so until the owner happens to claim successfully. No event distinguishes this from a level-up where the hook ran cleanly.

RECOMMENDATION

No code change is possible. Detect it off-chain: after every level change on the core, compare each of the key's `getStakes(nftId)[i].rate` values against `effectiveRate(nftId, tier)` and surface the divergence to the user with a one-click "claim to refresh your rate" action. Keeping the pool funded (ZS-01) removes the two balance-driven failure modes outright. For any successor: decouple the rate re-snapshot from the payout so a level-up always updates the rate even when the transfer leg cannot run, and emit an event from the core when the hook is caught.

RESOLUTION — ACKNOWLEDGED

Acknowledged. The mechanism is unchanged by the renouncement, but its consequence is worse: an operator could previously have refilled the pool through `depositTokens`, throttled depletion via `setBaseRewardRates` or `setReferralPercent`, or paused new stakes while the shortfall was resolved. Refilling survives — it needs no role, only an ERC-20 transfer — but the rate can never be reduced and new stakes can never be stopped.

ZS-14

The position is a bearer instrument attached to the key: an ERC-721 approval confers authority over the staked $ZEN, and a key sale is a race for the accrued reward

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Value binding / MEV
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:13-18 (the documented design), 131-134 (`_ownerOrRevert`) and its four uses at 196, 217, 232 and 291; 162-166 (`_accrued` measures from `lastClaimTimestamp` with no settlement at transfer), 253 and 267 (`_claimAll` pays `core.ownerOf(nftId)`)

DESCRIPTION

Authorisation for every value-moving function resolves through `core.ownerOf(nftId) == msg.sender`, and the contract holds no per-wallet record whatsoever. Whoever holds the key at call time controls 100% of the principal and 100% of the accrued reward. This is the contract's documented, intended design (lines 13-18) — positions are meant to travel with the key — but it has two consequences worth stating explicitly to users.

(a) Approval scope. `ZenFiNftKey` does not override any ERC-721 approval or transfer entrypoint, so a single `setApprovalForAll` — the most commonly phished signature in the ecosystem — hands an attacker the key and with it an arbitrary `$ZEN` balance of up to the 1,000,000 ZEN `maxStake`, with no cap, no delay and no separate consent step on the staking side. The renouncement provides one genuine partial mitigation here: because early withdrawal can never be enabled (ZS-11), an attacker who takes a key with an **unmatured** position cannot strip the principal ahead of maturity — though they can claim the accrued reward immediately, and take everything once the term ends.

(b) Sale timing. There is no settlement or snapshot at transfer time: whoever is `ownerOf` when a claim executes receives the entire accrual for that window, including the part that accrued under the previous owner. Both sides of a key sale can race in the mempool, and for a position already past maturity the race is over the **principal** as well, since a seller can front-run a marketplace fill with `withdraw`. The referral routing, by contrast, is clean and unaffected: `getUplineChain` resolves through the immutable `sponsorOf` genealogy, so a sale never redirects the sponsor lines.

SCENARIO / IMPACT

A holder with a level-9 key stakes 200,000 ZEN on the 1-year tier. On day 40 they sign a `setApprovalForAll` on a spoofed marketplace page. The attacker transfers the key into a fresh, key-less wallet — which `ZenFiNftKey`'s max-one-key rule permits — and is now `core.ownerOf(nftId)`. They can call `claimAllRewards` immediately and take the day-40 accrual, and the full 200,000 ZEN principal on day 365. The victim has no recourse and cannot exit ahead of them, because early withdrawal is permanently disabled.

Separately: a seller lists a key carrying a matured position, sees the buyer's fill in the mempool, and front-runs it with `withdraw` — the buyer receives an emptied key.

RECOMMENDATION

Treat the key as a bearer instrument for the entire staked balance and say so in the UI. Warn prominently that granting **any** NFT approval on a key also grants the staked `$ZEN`; encourage `setApprovalForAll(false)` hygiene and hardware wallets; and advise against routing a key through a marketplace while a position is open. Any OTC or marketplace sale of a key with an open position should be settled through an escrow that calls `claimAllRewards` in the same transaction as the transfer, so the accrual is realised deterministically to the agreed party; at minimum, surface per-position `calculateReward` and `lastClaimTimestamp` in listings so buyers can price the race. For any successor: record the staker's address alongside the `nftId`, or require an explicit opt-in before a transfer moves an open position.

RESOLUTION — ACKNOWLEDGED

Acknowledged as documented intended design (lines 13-18), reported because the **scope of an NFT approval** is not obvious to users and because the renouncement changed the risk profile in both directions: an unmatured position can no longer be stripped early by anyone (including an attacker), and equally no operator can freeze a compromised key's position.

ZS-15

Accrual has no maturity cap: a matured position keeps earning at its snapshotted rate indefinitely

MINOR

SEVERITY	MINOR
CONTRACT	ZenReferralStaking.sol
CATEGORY	Arithmetic / economics
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:162-166 (`_accrued` measures `block.timestamp - lastClaimTimestamp`), 206 (`startTime` recorded), 295 (`startTime` is read in exactly one place, only to decide whether a withdrawal is early), 300 (`withdraw` accrues with no maturity clamp)

DESCRIPTION

`_accrued` measures elapsed time from `lastClaimTimestamp` and multiplies by the snapshotted rate. `startTime` is read in exactly one place — line 295, purely to classify a withdrawal as early or not — and **nothing clamps the accrual window to `startTime + lockPeriods[tier]`**. A 30-day-tier position left in place for five years accrues 150% of its principal; a 1-year-tier position at rate 114 accrues 570%.

The lock period is therefore a **minimum holding time, not a maximum earning window**, and the product is in effect a perpetual annuity with a lock-up rather than a fixed-term instrument. This also settles a question worth recording: the reward earned between the last claim and the end of the lock is **not** dropped — `withdraw` accrues right up to `block.timestamp`, maturity included. The defect is the opposite one, that accrual does not stop.

Whether this is intended is a product question, not a code question, but it materially changes the reserve calculation behind ZS-01: the liability of an open position is not "one term of APR" but "APR × actual dwell time", which no one can bound in advance.

SCENARIO / IMPACT

A level-12 key stakes 200,000 ZEN on the 1-year tier and simply leaves it in place. Year one costs the pool 200,000 × 1.14 × 1.15 = 262,200 ZEN including the referral surcharge. Year two costs the same again, and year three likewise. After three years that single position has drawn 786,600 ZEN — against a contract that holds 1,215,921 ZEN today — and the 200,000 ZEN principal is still owed on top. Under the intuitive reading of a "1-year lock" the liability would have been one year of interest.

RECOMMENDATION

Model the reward budget as funding a **perpetual** obligation rather than a fixed term: the reserve requirement is APR × expected dwell time, and the pool must be topped up on a standing schedule rather than once per nominal term (see ZS-01). Decide deliberately whether a perpetual annuity is the intended product and describe it accurately in user-facing material — "1-year lock" is easily read as "one year of interest". For any successor: clamp the accrual window to `startTime + lockPeriods[tier] * 1 days` so a matured position stops earning and must be re-staked.

RESOLUTION — ACKNOWLEDGED

Acknowledged and permanent. Before the renouncement the exposure was bounded by the admin's ability to cut the tier rate via `setBaseRewardRates` — existing positions would have picked up the lower rate on their next claim through the re-snapshot at 223/260 — and by `pause` to stop new stakes. Both levers are gone.

ZS-16

`withdraw` pays the final accrued reward without calling `_payReferral` and without emitting `RewardClaimed` INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Accounting / observability
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:290-322 (`withdraw` , reward computed at 300, paid at 319, only `StakeWithdrawn` emitted at 321) against 216-229 (`claimReward` , referral at 226 and `RewardClaimed` at 228) and 252-270 (`_claimAll` , referral at 268 and `RewardClaimed` at 263); header NatSpec at 31-35

DESCRIPTION

`withdraw` computes `rewardAmount` at line 300 and pays it to the owner at line 319 bundled with the principal. That is a reward payment by every definition, yet unlike `claimReward` and `_claimAll` it never calls `_payReferral` and never emits `RewardClaimed` — only `StakeWithdrawn` , which carries the amount in a non-indexed field.

Two consequences follow. **Sponsor income:** the header NatSpec at lines 31-35 states that *every* claim pays an extra referral share to the owner's sponsors, yet the most natural user behaviour — stake, wait out the lock, withdraw once — produces exactly zero `ReferralPaid` events. For a 1-year position never intermediately claimed, the whole reward is delivered through line 319 and the three upline lines receive nothing for the entire year. The referral programme is therefore silently opt-out at the downline's discretion. Note the direction of the effect on solvency is **favourable**: this path costs the pool principal + reward rather than 1.15 × reward.

Accounting: any lifetime-reward or referral-turnover figure keyed only on `RewardClaimed` under-reports. ZenFi's own indexer already compensates — it writes a separate row from `StakeWithdrawn.rewardAmount` precisely for this reason — so this is a hazard for **third-party integrators**, not a live under-reporting bug in this system.

SCENARIO / IMPACT

A staker with three real sponsors stakes 100,000 ZEN on the 1-year tier and, being already at the maximum key level, never triggers the core's `claimFor` hook and never claims manually. On day 365 they call `withdraw`: line 300 accrues 114,000 ZEN, line 319 transfers 214,000 ZEN, and line 321 emits `StakeWithdrawn`. Their three sponsors, who under the documented model were owed 3 × 5% × 114,000 = 17,100 ZEN, receive nothing and no `ReferralPaid` event is emitted.

RECOMMENDATION

Correct the NatSpec claim wherever it is reproduced in user-facing material: the referral share is paid on **claims**, not on the reward settled inside a withdrawal. Sponsors should understand that a downline who never claims generates no referral income. Any third-party integrator computing staking income must add `StakeWithdrawn.rewardAmount` to `RewardClaimed`, as ZenFi's own indexer does. For any successor: call `_payReferral` and emit `RewardClaimed` in `withdraw` so the final tranche is treated identically to every other claim.

RESOLUTION — ACKNOWLEDGED

Acknowledged and permanent; no administrative function ever had any bearing on this path. Reported as Informational because the protocol's own accounting already handles it and the solvency effect is favourable.

ZS-17

`withdraw`

never checks that the position is open, and zeroing the slot discards its tier — a repeat call is stopped only by the token's zero-transfer guard

INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Insufficient validation
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:290-322, specifically the missing `amount > 0` guard after 292, the `lockPeriods[userStake.tier]` reads at 293 and 296 against the tier zeroed at 313, and the transfer at 319; also the unchecked `stakeIndex` at 218, 292; cross-contract: `Zen.sol` :334 (`require(amount > 0)`)

DESCRIPTION

After a withdrawal the struct is zeroed in place (310-314) but the array element remains. Calling `withdraw(nftId, sameIndex)` again passes **every** check in the function: `_ownerOrRevert` succeeds; line 293 reads `lockPeriods[0]` — because line 313 set `tier = 0` — which is 30 and therefore non-zero; line 295 computes `elapsedSeconds` from a zero `startTime`, an enormous value, so line 296 classifies it as not-early and line 297 passes even with early withdrawal disabled; every amount is zero and `nftTotalStaked[nftId] -= 0` is a no-op. Execution reaches `safeTransfer(owner, 0)` at line 319 and is rejected **only** because `$ZEN` refuses zero-value transfers.

So the invariant is enforced by an external token's implementation detail rather than by this contract's own code. Against an ERC-20 that permits zero-value transfers — the overwhelming majority — the same call would succeed and emit a fully-formed but entirely spurious `StakeWithdrawn`. That variant is now unreachable forever, because `setStakingToken` is permanently uncallable and the token is pinned to `$ZEN` (ZS-07).

The more substantive half is the tier zeroing at line 313: it destroys the historical tier of every closed position, so `getStakes(nftId)` cannot tell a caller which tier a settled position belonged to and the `StakeWithdrawn` event does not carry it either. Separately, an out-of-range `stakeIndex` reverts with a bare `Panic(0x32)` rather than a descriptive error.

SCENARIO / IMPACT

A staker withdraws index 0 at maturity; lines 310-314 zero the struct. A UI retry, or a bot reacting to `StakeWithdrawn`, calls `withdraw(nftId, 0)` again. Every `require` passes and control reaches line 319 with a zero payout; `$ZEN` reverts with *"Transfer amount must be greater than zero"* — a message that gives the user no indication that the real problem is that the position is already closed.

RECOMMENDATION

Off-chain consumers should treat any `StakeWithdrawn` with all-zero amounts as spurious, and should reconstruct a closed position's tier from its `StakeCreated` event rather than from `getStakes`. For any successor: add `require(userStake.amount > 0, "Position closed")` immediately after loading the struct, preserve `tier` when settling (or swap-and-pop the entry), guard the final transfer with `if (payout > 0)`, and bounds-check `stakeIndex` with a named error.

RESOLUTION — ACKNOWLEDGED

Acknowledged, with **no impact under the frozen configuration**. The dangerous variant depended on `setStakingToken` repointing at a token that permits zero-value transfers; that function is permanently uncallable, so `$ZEN`'s own guard now supplies the missing check for the life of the contract.

ZS-18

The constructor validates less than the setters it feeds, and `minStake`, `maxStake` and `lockPeriods` never had setters at all INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Insufficient validation
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:94-126 (constructor: address checks at 105-107, equal-length check at 108-112, no value validation), against 358-369 (`setEarlyWithdrawalPenalties`, which does cap each entry at `BPS`); `stake` at 195-212 versus `withdraw`'s `lockPeriods[tier] > 0` requirement at 293

DESCRIPTION

The constructor checks that the three addresses are non-zero and that the three arrays are the same length, and validates nothing else. It does not require the arrays to be non-empty, does not cap any reward rate, does not require `lockPeriods` entries to be non-zero, does not cap the early-withdrawal penalties the way `setEarlyWithdrawalPenalties` later does, and does not require `minStake <= maxStake`. A deployment with empty arrays would have accepted no tier at all; one with `minStake > maxStake` would have rejected every stake; one with a zero-length lock period would have accepted deposits that `withdraw` then refuses at line 293 with *"Staking period not set"* — `stake` has no matching check, so the two functions disagree about what a valid tier is.

Because `minStake`, `maxStake` and `lockPeriods` have **no setters anywhere in the contract**, a constructor mistake in any of them would have been unfixable from the moment of deployment, long before the renouncement. This is recorded for completeness: the deployed values were read directly from the chain and are all sane and mutually consistent.

SCENARIO / IMPACT

Historical / counterfactual only. Had the deploy script passed `lockPeriods = [0, 90, 180, 365]`, tier 0 would have accepted deposits at line 200 (which validates only `_tier < baseRewardRates.length`) and then rejected every withdrawal at line 293, permanently locking those positions with no administrative path to correct the array.

RECOMMENDATION

For any successor: validate constructor parameters at least as strictly as the setters that later modify them — non-empty arrays, non-zero lock periods, a rate ceiling, penalties capped below `BPS`, and `minStake <= maxStake` — and mirror `withdraw`'s `lockPeriods[tier] > 0` requirement in `stake` so the two functions cannot disagree. No action is possible or required on this deployment.

RESOLUTION — ACKNOWLEDGED

Acknowledged. No live risk: every deployed value was verified on-chain — `[30, 40, 50, 100]` %, `[30, 90, 180, 365]` days, `[3000, 4500, 6000, 7500]` bps, `minStake` 1 ZEN, `maxStake` 1,000,000 ZEN — and all are consistent and within sane bounds. They are now permanent.

ZS-19

With `depositTokens` permanently uncallable, every future top-up is a bare ERC-20 transfer that emits no `TokensDeposited` event INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Observability (created by the renouncement)
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:87 (the `TokensDeposited` declaration, now unreachable), 326-329 (`depositTokens`, its only emitter), 78 (`nftTotalStaked`, the only accounting state), 199 (its only read), and the outflows at 225, 267, 283 and 319

DESCRIPTION

Because the audit needed to establish that the pool can still be funded after the renouncement, this was checked exhaustively rather than assumed. The contract declares no `totalStaked`, `rewardPool` or `totalDeposited` accumulator. The only accounting state is the per-key `nftTotalStaked`, written at 204 and 318 and read only by the `maxStake` guard at 199 — it never gates a payout. `depositTokens` itself performs a bare `safeTransferFrom` and emits an event; it updates no state. Every outflow is an unconditional `safeTransfer` whose success depends purely on `balanceOf(address(this))`.

Conclusion: a plain `ZEN.transfer(staking, amount)` increases payable capacity in exactly the same way `depositTokens` did, and requires no role. This is also how the pool was originally funded at deployment. The permanent loss of `depositTokens` therefore costs nothing functionally.

What it does cost is attribution. `TokensDeposited` can never be emitted again, so a top-up leaves only a generic ERC-20 `Transfer` log with nothing distinguishing it from any other `$ZEN` movement. That matters because pool solvency is the load-bearing control behind ZS-01, and any monitor keyed on `TokensDeposited` will now report zero deposits forever. It is worth noting that such a monitor was never complete — a raw transfer has **always** funded the pool without emitting the event — but the renouncement removed the last attributable funding path.

SCENARIO / IMPACT

The operator decides to add 500,000 ZEN to the reward pool. `depositTokens(500_000e18)` reverts with `AccessControlUnauthorizedAccount`. Instead any wallet calls `ZEN.transfer(0x1b7418DA…, 500_000e18)`; `$ZEN`'s wallet-to-wallet branch is untaxed, the balance rises by exactly 500,000, and the very next claim or withdrawal draws against it with no further step — no state to sync, no flag to set. The only trace is a `Transfer` log in `$ZEN`.

RECOMMENDATION

Publish the funding procedure explicitly: top-ups are made by a plain `$ZEN` transfer to `0x1b7418DA30b5C9eD15C228A992ba3d4a1cEE8B3C`, are permissionless, and are irreversible. Re-key all pool monitoring onto `$ZEN` `Transfer` events with `to == staking` rather than `TokensDeposited`, and publish the funding wallet so top-ups remain attributable off-chain. Build the public solvency figure the contract cannot express itself: balance minus open principal derived from `StakeCreated` / `StakeWithdrawn`. For any successor: make the deposit wrapper permissionless — funding a pool needs no privilege — so the event survives any future renouncement.

RESOLUTION — ACKNOWLEDGED

Acknowledged; functionally benign and confirmed line by line. `depositTokens` was a convenience wrapper with no accounting side effect, and the permissionless plain transfer is a complete substitute. The only real consequences are the loss of the attributable event trail, addressed by the monitoring change above, and the irreversibility of a top-up, covered by ZS-12.

ZS-20

A referral line terminating at the treasury is dropped rather than compressed, and nothing is emitted for an unpaid line

INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Logical issue / observability
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:272-274 (the documented intent), 278 (`getUplineChain` with `maxDepth = 3`), 279 (`core.treasury()`), 282 (the `continue`), 284 (`ReferralPaid` , emitted only on a successful payment); cross-contract: `ZenFiNftKey.sol` :1040-1045 (the walk terminates at the root)

DESCRIPTION

Line 282 uses `continue`, so a line whose owner is the treasury — or, unreachably, the zero address — is simply not paid; the contract does not walk further up to substitute a real sponsor. Because the root key is the only one with no sponsor and `getUplineChain` stops there, the treasury can only ever occupy the **last** slot of the returned array, so the skip always costs the tail of the chain. Combined with the fact that `getUplineChain` returns a variable-length array shorter than three for shallow keys, the number of paid lines silently ranges from zero to three depending purely on registration depth.

This is the intended economics — the project's own test asserts that the treasury is excluded — and it is pool-positive, so no funds are lost. The reportable part is observability: `ReferralPaid` fires only on a successful transfer and there is no counterpart event for a skipped or empty line, so the absence of a `ReferralPaid` is indistinguishable across three different causes — a chain shorter than three, a treasury terminal skipped at line 282, and a per-sponsor amount that floored to zero and returned early at line 277.

SCENARIO / IMPACT

A key minted directly under the root has an upline chain of exactly one entry: the treasury. `_payReferral` skips it at line 282 and returns having paid nobody and emitted nothing at all. A key five hops down pays the full 15%. On a 100,000 ZEN 1-year-tier position at level 5, the first claim costs the pool 100,000 ZEN per year and the second 115,000 ZEN — a 15,000 ZEN/year difference that an indexer cannot attribute, because both cases look identical in the logs except for the presence or absence of `ReferralPaid` rows, which is also exactly what a short chain looks like.

RECOMMENDATION

Reconcile pool outflow off-chain by reconstructing each claimer's `sponsorOf` chain independently rather than inferring it from the absence of events, and expose the expected number of paid lines in the UI so a user is not surprised that their upline earns nothing. For any successor: emit an explicit `ReferralSkipped(fromNftId, lineIndex, reason)` for every unpaid line so outflow reconciles line by line, drop the unreachable zero-address branch, and decide deliberately whether a treasury terminal should collapse the line or compress to the next real ancestor.

RESOLUTION — ACKNOWLEDGED

Acknowledged as intended, test-pinned behaviour with no fund loss. The skip is unconditional code with no administrative parameter behind it, so the renouncement neither neutralises nor worsens it; the only indirect effect is that `setReferralPercent` being dead removes the last way to make the per-line asymmetry economically smaller.

ZS-21

`stake`

credits the requested amount with no received-amount check — verified harmless for $ZEN, and now permanently pinned

INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Token integration
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:195-212 (`stake` : state written at 204-207, transfer at 209); cross-contract: `Zen.sol` :343-345 (pre-launch LP lock), 348-361 (tax gating), 523 (`uniswapV2Pair` assigned once, no setter); contrast `ZenFiNftKey.sol` :722-725 (a balance-delta check with an explicit revert)

DESCRIPTION

`stake` increments `nftTotalStaked` at line 204 and records `Stake(_amount, …)` at 205-207 using the caller-supplied `_amount` , then calls `safeTransferFrom` at line 209. The checks-effects-interactions ordering is correct — all state is written before the external call — but the consequence is that the contract credits what was **requested**, not what **arrived**. There is no balance-before/balance-after measurement, which is notable because the sibling core does exactly the opposite and reverts on any mismatch, explicitly to reject fee-on-transfer and rebasing tokens.

The hazard does not exist for the live token, and the argument is stronger than the fee-exclusion flag alone. `$ZEN` charges tax only when one side of the transfer is the Uniswap pair — a buy or a sell — and every other transfer falls through untaxed; `uniswapV2Pair` is assigned exactly once, in `openTrading` , and has no setter. A `safeTransferFrom` from a wallet to the staking contract is neither, so no tax can apply **regardless** of the fee-exclusion flag, and the contract's balance rises by exactly the credited amount. The fee exclusion is belt-and-braces, not the load-bearing guarantee.

One evidentiary caveat: the project's test that asserts exact crediting runs against a plain mock ERC-20, not against `$ZEN` , so the `$ZEN` behaviour is established here by code reading rather than by that test.

SCENARIO / IMPACT

Not an exploit — the refutation, concretely. A level-5 key calls `stake(nftId, 1000e18, 0)` . Line 204 records 1000e18 and line 209 pulls 1000e18 `$ZEN` ; neither side of the transfer is the pair, so the entire tax and swap-back block is skipped and the contract's balance rises by exactly 1000e18. Credit and receipt match to the wei.

RECOMMENDATION

Record the invariant so it survives personnel changes: this contract is only safe with a token that transfers exactly the requested amount, and it verifies nothing itself. Anyone evaluating a future ZenFi staking contract should require the balance-delta pattern used by `ZenFiNftKey._collectPayment` in `stake` — it costs two storage reads and removes an entire class of integration risk. No action is required for `$ZEN`.

RESOLUTION — ACKNOWLEDGED

Permanently neutralised in the strongest sense. The residual risk was never `$ZEN` but `setStakingToken` (ZS-07), which could have repointed the contract at a fee-on-transfer or rebasing token and turned the unmeasured credit at 204-207 into systematic over-crediting paid out of other stakers' principal. That function has no possible caller, so the token is pinned to `$ZEN` and the guarantee above holds for the life of the contract.

ZS-22

Verification notes: the accrual arithmetic is correct; the referral loop round-trips the sponsor chain through addresses; the early-withdrawal branch charged an undocumented second penalty and is now dead code

INFORMATIONAL

SEVERITY	INFORMATIONAL
CONTRACT	ZenReferralStaking.sol
CATEGORY	Code quality
STATUS	ACKNOWLEDGED
LOCATION	contracts/staking/ZenReferralStaking.sol:162-166 (`_accrued`), 278-284 (`_payReferral` 's address round-trip), 303-307 (the early-withdrawal penalty branch, second penalty at 306), 365 (the `<= BPS` cap)

DESCRIPTION

Three observations that carry no security impact, recorded because they were the output of the unit-by-unit verification.

(a) The accrual arithmetic is sound — no issue found. Line 164 converts the whole-percent rate to basis points and line 165 divides by `BPS * 365 days` ; the two scalings cancel to `amount * elapsed * rate / (100 * 365 days)` , which over one year yields exactly `rate` percent of principal. Verified numerically at 100% and 114%. No overflow is reachable — the numerator is bounded by `maxStake` × elapsed × the maximum basis rate, dozens of orders of magnitude below the `uint256` ceiling — and no dust truncation is reachable either: at the live `minStake` and lowest rate, one second of accrual is far above the truncation floor, so `_accrued` never rounds a legitimate reward to zero. The only stylistic note is that the redundant `× 100 / BPS` pair fixes the APR at whole-percent granularity, which is now permanent.

(b) The sponsor chain is round-tripped through addresses. `_payReferral` receives the staked `nftId` but derives the chain from the owner **address** at line 278, which sends the core back through `nftIdOf` to recover an id the caller already had; it then discards the ids the core just walked and pays a second cross-contract call per iteration at line 284, purely to fill an event field. That is six cross-contract reads per claim where four would do — roughly 10k gas of redundant lookups — and it makes the referral leg's correctness rest on the core's one-key-per-wallet invariant, which this contract never asserts. The invariant does hold in the deployed core, so the behaviour is correct as deployed.

(c) The early-withdrawal branch is now dead code, and while live it was mis-documented. Line 305 charges the documented principal penalty, and line 306 then charges the **same** `penaltyBps` a second time against the accrued reward — nothing in the NatSpec mentions this, and the only prose on the subject describes the principal penalty alone. At the configured tier-3 penalty an early exit would have cost 75% of principal **and** 75% of accrued interest. The cap at line 365 is also off by one: at exactly `BPS` the payout becomes a zero-value transfer, which `$ZEN` rejects, so early withdrawal in that tier would always have reverted; the correct cap is `< BPS` . Both points are now purely historical.

SCENARIO / IMPACT

Not exploits. For (a), a 1,000 ZEN position at rate 114 held for one year yields exactly 1,140 ZEN, and one second yields roughly 3.6×10^{13} wei — non-zero, so per-second claiming works and loses under one wei to truncation per call. For (c), had early withdrawal ever been enabled, a user exiting a 100,000 ZEN tier-3 position after 200 days would have received 25,000 ZEN of principal plus 25% of the accrued interest, roughly 40,600 ZEN in total, whereas the documented model implies 87,500 ZEN.

RECOMMENDATION

For any successor: write the accrual as `amount * elapsed * rate / (100 * 365 days)` and name the field `ratePercent`, or move to basis points end-to-end so fractional APRs become expressible; add an id-keyed upline view to the core and consume that instead of the address round-trip, binding the payout to the staked key rather than to the claimer's address; and either document the reward haircut on early exit explicitly or remove it, tightening the penalty cap to `< BPS`. On the live contract, remove the early-withdrawal penalty schedule from user-facing material — the values returned by `getStakingInfo()` describe a route that does not exist.

RESOLUTION — ACKNOWLEDGED

Acknowledged; no action possible or required. (a) is a verification note confirming correctness rather than a defect. (b) involves no administrative surface and `core` was already `immutable`. (c) is permanently unreachable: `toggleEarlyWithdrawal` and `setEarlyWithdrawalPenalties` can never be called, so `isEarlyWithdrawalEnabled` is fixed at false and lines 303-307 can never execute.

Conclusion

`ZenReferralStaking` is a small, disciplined contract that does what its documentation says. Authorisation is a single consistent `ownerOf` check; checks-effects-interactions ordering is correct on every value-moving path and each carries a reentrancy guard; the accrual arithmetic was verified unit-by-unit to produce exactly the advertised annual percentage with no reachable overflow or truncation; and the `$ZEN` integration is safe by construction. No Critical finding was identified and no externally-exploitable path to user funds exists.

The headline governance fact is verified, not asserted: the complete nine-event role history shows `ADMIN_ROLE` and `DEFAULT_ADMIN_ROLE` with no holder and no possible future holder. That permanently closes this contract's centralization surface — the admin could previously have moved the entire balance out with no reserve check and no event (ZS-06), re-denominated every open position into another token (ZS-07), or rewritten the rates and penalties governing open positions (ZS-08). All three are now impossible for anyone. This is a materially stronger guarantee than a multisig or a timelock, and it is the right frame for reading this report.

The same permanence is the source of the remaining risk, and the report states it plainly. The single **Major** finding (ZS-01) is that principal and the reward budget share one unreserved balance with no solvency invariant, no global stake cap and — now — no throttle: at the top tier the pool pays out up to 131% of principal per year once the 15% referral surcharge is counted, accrual never stops at maturity (ZS-15), and a shortfall would surface as failed claims and blocked withdrawals in first-come-first-served order (ZS-02, ZS-03). Measured on-chain, the contract holds 1,215,921 ZEN against 223,387 ZEN of open principal, so the free buffer covers even a worst-case reading of the current book for about 3.4 years — the pool is solvent today, and the risk is prospective. Crucially, the remedy survives the renouncement: **anyone can fund the pool by a plain $ZEN transfer to the contract**, because `depositTokens` was only an event-emitting wrapper and no payout path consults an internal ledger.

Our recommendations are therefore operational, since no code change is possible. Publish and monitor a solvency figure the contract cannot express itself — balance minus open principal, reconstructed from `StakeCreated` / `StakeWithdrawn` — and pre-fund each accepted position for its full term rather than topping up after a failure is reported. Budget rewards at 115% of nominal APR, since the referral surcharge is unavoidable and self-capturable (ZS-04). Surface three product facts prominently before a user stakes: there is no early exit at any price (ZS-11), the staked balance follows the NFT key and any approval on that key (ZS-14), and there is no administrator who can intervene. Finally, extend the renouncement disclosure to `ZenFiNftKey`, whose administration is not renounced and on which this contract's economics still depend (ZS-05). With those controls in place the contract is in good shape for continued production use.

Appendix A — Severity Classification

SEVERITY	DEFINITION
CRITICAL	Vulnerabilities leading to loss of funds, asset theft, or complete contract compromise, exploitable by an external party. Must be fixed before deployment.
MAJOR	Issues that can cause significant fund loss or break core functionality under realistic conditions, or grant unauthorized privileged access.
MEDIUM	Issues that can lead to conditional loss/locking of funds, replay across contexts, or material deviation from intended behavior given specific preconditions.
MINOR	Limited-impact issues, robustness/best-practice gaps, or issues requiring a trusted-actor mistake to manifest.
INFORMATIONAL	Code style, gas optimization, documentation, and defense-in-depth suggestions with no direct security impact.

Appendix B — Finding Categories

CATEGORY	MEANING
Access Control / Centralization	Powers concentrated in privileged roles that can affect user funds or contract behavior.
Signature Verification	Issues in the signing scheme, message construction, or replay protection.
Reentrancy	Unsafe external-call ordering or missing reentrancy protection.
Arithmetic / Accounting	Rounding, overflow/underflow, or ledger-consistency errors.
Oracle / Pricing	Price-feed correctness, staleness, and manipulation resistance.
Logical Issue	Flaws in the business/protocol logic.
Denial of Service	Conditions that can render functionality unusable.
Token Integration	ERC-20/721 compatibility, fee-on-transfer, and reward-token handling.
Insufficient Validation	Missing or inadequate input validation.
Code Quality	Style, consistency, gas, and documentation.

Disclaimer