smart state

Web3 security easier than ever

SOLQueef

D8QrznnS1uB4LCVJaA7WzF8SioSexgcevEWj7sg1vxrp

Security Audit Report

Ver. 1.1
November 14, 2025

smartstate.tech

1. Report Summary

The project **Queef Coin ($SOLQUEEF)** is a meme-driven Solana ecosystem that turns viral culture into a gamified, carbon-positive economy where community quests finance verifiable environmental initiatives. Through Anchor presale and vesting contracts, oracle-based SOL and stablecoin payments, and QQDAO governance, it ties transparent on-chain capital flows to impact-focused meme campaigns led by $SOLQUEEF holders.

Discoveries Summary

Issue Severity	Discovered
INFO	5
LOW	1
MEDIUM	3
HIGH	0
CRITICAL	0
Total:	9

Conclusion

The project implements a substantial body of novel Solana smart-contract code for a token presale, combining SOL and USDC/USDT purchase flows with oracle-based pricing and configurable vesting. Despite this multi-component architecture, technical analysis has found no fundamental flaws or reasons to conclude that the system could not operate successfully under its intended operating conditions.

2. System Review

The security model of the Queef Coin presale system relies primarily on the correctness and integrity of its Solana application-layer logic and its interaction with price oracles, canonical stablecoin mints, and treasury wallets. This System Review section focuses on that application-layer logic, splitting it into the following components:

1. Token Presale: Capital Inflow

The presale mechanism is implemented as an Anchor program on Solana, exposing instructions for configuration (`initialize`, `set_price`, `set_live`, `set_vesting_config`, `set_vestingtime`) and for user participation (`buy`, `buy_with_stable_coin`). Global parameters such as the admin address, vault wallet, token prices, vesting schedule, and aggregate counters are stored in a single `GlobalState` account, while each participant's allocation and purchase history reside in a per-wallet `UserState` account. This makes presale configuration and user positions explicit, queryable, and verifiable on-chain.

For SOL contributions via `buy`, the program receives lamports from the user, queries a fixed Pyth `SOL/USD` feed, and derives the user's token allocation using the oracle price and the admin-set token price, with strict checks on positive prices, non-zero amounts, and bounded oracle age. Global sold-amount counters are updated with checked arithmetic, and SOL is forwarded directly to a designated vault wallet whose address is configured in `GlobalState`. For stablecoin contributions via buy_with_stable_coin, the program enforces canonical USDC/USDT mints, scales 6-decimal stablecoin amounts into a 9-decimal internal base, and uses the configured stablecoin price to compute a deterministic token allocation. USDC/USDT then flows from the user's associated token account to the vault's associated token account via CPI to the SPL Token program. In both cases, the presale must be live before funds are accepted, tokens are never minted by this program, and all user entitlements are represented as state in `UserState` against a separate PDA-owned token vault.

2. Vesting and Token Distribution

The vesting subsystem governs time-based release of presale allocations, with global parameters (`vesting_percentage`, `max_vesting_periods`, `vesting_duration`, `vesting_start`) set in `GlobalState` and per-user progress tracked in `UserState` (`tokens`, `total_vested_tokens`, `last_vesting_date`). Subsequent purchases increment a user's total allocation without resetting previously vested amounts, so vesting accrues monotonically across the full presale position. Configuration instructions guard against nonsensical schedules by requiring strictly positive percentages, durations, and period counts, and by disallowing per-period vesting amounts that would round to zero.

Token release is handled by the `claim_vesting` instruction, which combines authorization, controlled token movement, PDA signing, and state-based logic. Authorization is enforced on-chain with `require!(is_user || is_admin, PresaleError::UnauthorizedClaim)`, so only the beneficiary or the admin can initiate a claim. Token transfers occur solely between the designated vault's presale token ATA and the user's ATA, with the vault address validated against `global_state.vault` and the SPL Token authority set to a deterministic program-derived address (`pdaauthority`) whose seeds and bump are known and verified at runtime; only this PDA signer can approve outgoing transfers. The claimable amount is derived from elapsed time since `last_vesting_date`, the configured `vesting_duration`, and the remaining number of periods up to `max_vesting_periods`, ensuring that premature, repeated, or over-large claims are rejected. All invariants are enforced via explicit `require!` checks over both global configuration and user state, so vesting cannot be accelerated or redirected without either admin-level state changes or a governed upgrade of the program itself.

3. Periphery

This section describes the infrastructural components that connect the presale smart contract to external services, governance, and the broader Queef Coin ecosystem. Together, they form the coordination layer that ties meme-driven campaigns and environmental initiatives to transparent on-chain capital flows, while relying on the contract's invariant-enforcing logic for correctness. Key elements include the Pyth oracle integration that supplies SOL/USD prices, the SOL and USDC/USDT treasury vaults that collect presale proceeds, the PDA-owned presale token vault that backs vesting claims, and the admin and upgrade authorities (EOA, multisig, or QQDAO) that control configuration and, ultimately, program mutability.

These components are essential to the correct execution of presale and vesting, even though many of their guarantees (e.g., who controls the vault, who holds the upgrade authority, how proceeds are deployed) are governance or operations questions rather than purely code-level ones. Oracle operators must publish timely and honest prices; vaults must be secured with appropriate multisig or DAO policies; and the program's upgrade authority must be locked or placed under community control once audited code is deployed. On top of this base, frontend applications, the quest engine, and QQDAO governance consume on-chain presale and vesting data to manage user experience, distribute $SOLQUEEF rewards, and route collected funds into verifiable environmental projects. The following modules are included in this group:

- 3.1 Pyth Price Oracle and Feed Management
- 3.2 SOL and Stablecoin Treasury Vaults
- 3.3 PDA-Owned Presale Token Vault and Claim Flow
- 3.4 Admin, Upgrade Authority, Frontend, and Quest/DAO Integration

3. Audit Scope

The Queef Coin ($SOLQUEEF) presale system may be classified as a single-program, oracle-integrated Solana application that manages capital inflows and time-based token distribution, with security anchored in on-chain state machines, PDA-mediated token custody, and externally governed treasuries. This classification allows the application of well-established assessment methodologies for Solana programs that depend on authenticated price feeds and admin-controlled configuration, with particular attention to trust in the admin and upgrade authorities, correctness of PDA derivations, oracle usage, and SPL Token transfer flows.

Within this context, the audit covers the complete Anchor presale program and its directly associated components: global and per-user state accounts, SOL and USDC/USDT purchase paths, Pyth price integration, vesting and claim logic, admin configuration instructions, and the PDA-owned token vault through which vesting payouts are executed. Functional review and architectural reasoning are treated as a single, integrated track: for each instruction and execution path, the analysis jointly examines behavioural correctness (state transitions, arithmetic safety, access control, and vesting schedule enforcement) and systemic soundness (admin powers, reliance on external price and treasury infrastructure, upgradeability, and user exit guarantees). The goal is to determine whether, under the stated trust assumptions, the program can safely account for contributions, prevent unauthorized token movement or premature vesting, and support the project’s governance and environmental-impact objectives.

For detailed information regarding the specific contracts, accounts, and third-party dependencies included within the scope of this audit, as well as their concrete deployment identifiers, please refer to the **Verification Checksums** section.

I. Functional Audit

1. Discoveries:

1. Unbounded Administrative Control over User Allocations	MEDIUM

Description: The `set_userdata` instruction gives the admin direct write access to a user's recorded allocation and vesting progress, without any restriction. The admin can arbitrarily set `tokens`, `total_vested_tokens`, and `last_vesting_date` for any user:

```
  pub fn process_instruction(
      ctx: Context<Self>,
      token_amount: u64,
      vested_tokens: u64,
      last_vesting_date: i64,
  ) -> Result<()> {
      let user_state = &mut ctx.accounts.user_state;
      require!(vested_tokens <= token_amount,
PresaleError::InvalidUserData);
      user_state.tokens = token_amount;
      user_state.total_vested_tokens = vested_tokens;
      user_state.last_vesting_date = last_vesting_date;
      Ok(())
  }
```

While this is consistent with a centralized presale, it effectively allows the admin (EOA or multisig) to zero out users, gift allocations to preferred addresses, or fast-forward vesting by setting total_vested_tokens close to tokens. This is not a coding bug but a strong governance assumption; if users expect on-chain guarantees, the function should be time-limited, disabled after launch, or placed under DAO/multisig governance with transparent policies.

2. Mutable Treasury Vault Without On-chain Locking

MEDIUM

Description: The `set_vault_address` instruction permits the admin to change the treasury vault address at any time, with only a non-zero check on the new key:

```
  #[account(mut, seeds = [GLOBAL_SEED], bump)]
  pub global_state: Account<'info, GlobalState>,
  pub fn process_instruction(ctx: Context<Self>) -> Result<()> {
      let global_state = &mut ctx.accounts.global_state;
      require!(ctx.accounts.vault.key() != Pubkey::default(),
PresaleError::InvalidVaultAddress);
      global_state.vault = ctx.accounts.vault.key();
      Ok(())
  }
```

Because `buy` and `buy_with_stable_coin` route all incoming SOL and USDC/USDT directly to `global_state.vault`, the admin can silently redirect new funds to any wallet mid-presale. From a trust perspective, this means users must fully trust the admin not only at deployment time but throughout the lifetime of the sale.

3. Upgradeable Program Can Be Replaced After Deployment

MEDIUM

Description: At the Solana layer, the presale program remains upgradeable unless the deployer explicitly revokes the upgrade authority (e.g., `--new-upgrade-authority none`). This risk does not appear in the Rust code itself but in deployment configuration: an entity holding upgrade authority can redeploy arbitrary bytecode under the same program ID, retroactively changing presale rules, vesting behavior, access control, or even draining token vaults.

Because users typically identify contracts by program ID, not binary hash, this creates a powerful centralization point even if the current source code is audited and verified. The security model therefore assumes that the upgrade authority will be (a) transferred to a secure multisig or DAO and (b) eventually set to none once the presale logic is finalized and verified. Until that happens, the system's guarantees are "best-effort" and depend heavily on off-chain governance rather than immutable code.

4. Potential Use of Stale Pyth Prices in SOL Purchases

LOW

Description: The SOL purchase path reads Pyth pricing via `get_price_no_older_than` but allows price updates up to 600 seconds old:

```
  let price_update = &mut ctx.accounts.price_feed;
  let maximum_age: u64 = 600;
  let feed_id: [u8; 32] = get_feed_id_from_hex("0xef0d...b56d")?;
  let price = price_update.get_price_no_older_than(&Clock::get()?,
maximum_age, &feed_id)?;
  let asset_price = price.price;
  require!(asset_price > 0, PresaleError::InvalidPriceFeed);
```

While this protects against arbitrarily stale data, a 10-minute window is relatively wide for volatile markets: during sharp moves, users could buy at prices that deviate significantly from current spot, creating economic unfairness or exploitable arbitrage for fast actors.

This is not a correctness bug—the code behaves as written—but it is a parameter-level risk.

5. No On-chain Time or State Limits for Risky Admin Functions

INFO

Description: Several admin instructions—such as `set_userdata`, `set_vault_address`, `set_vesting_config`, and `set_vestingtime`—remain callable indefinitely, regardless of whether the presale is live or completed. Although the project intends to rely on social processes (multisig/DAO governance and clear policies) and possibly treat `is_live` as a de-facto freeze point, the code does not enforce any "admin lockdown" after launch.

As a result, even long after users have committed funds, the admin can adjust vesting parameters, re-point the vault, or rewrite user allocations.

6. No On-chain Enforcement of Presale Caps or Per-user Limits

INFO

Description: The program tracks aggregate and per-user volumes (`token_sold`, `solana_recieved`, `stable_coin_recieved`, `UserState.paid_sol`, `UserState.paid_usd`) but does not enforce any soft or hard caps:

```
global_state.token_sold = global_state
    .token_sold
    .checked_add(token_amounts)?;
user_state.paid_sol = user_state
    .paid_sol
    .checked_add(sol_amount)?;
```

There is no maximum total supply for the presale encoded in `GlobalState`, nor per-wallet or per-transaction limits to constrain whales or bots. This is not a security flaw in the narrow sense—the contract will not misaccount or overflow—but it means guarantees about maximum distribution are entirely off-chain.

7. Single Go-live Flag Does Not Provide Granular Emergency Controls

INFO

Description: The `is_live` flag in `GlobalState` is used to gate purchases:

```
if global_state.is_live == false {
    return Err(error!(PresaleError::PresaleNotStarted));
}
```

However, `is_live` is checked only in `buy` and `buy_with_stable_coin`; it does not affect `claim_vesting` or admin instructions like `set_userdata` and `set_vault_address`. In an emergency (e.g., suspected oracle issue or treasury compromise), the admin can immediately halt new buys but cannot, at the code level, prevent further vesting claims or risky admin actions without resorting to a full program upgrade.

This design is acceptable for a simple presale but offers limited incident-response granularity.

8. Limited On-chain Transparency Through Events and Structured Logging

INFO

Description: The program currently does not emit structured events (e.g., Anchor `#[event]`) for key actions such as purchases, vault changes, vesting claims, or admin reconfigurations. State transitions are observable by scanning accounts and transactions, but without explicit events, off-chain indexers, explorers, and analytics tools must reverse-engineer changes from account diffs, which is slower, more error-prone, and less transparent to end users.

For a project that emphasizes environmental impact and transparent fund flows, rich event emission (e.g., `PresalePurchased { user, amount_in, tokens_out }`, `VaultChanged { old_vault, new_vault }`, `VestingClaimed { user, amount, periods }`) would materially improve auditability and UX. While this is not a security vulnerability, it directly affects how easily the community and auditors can verify that presale operations match the stated rules.

9. Vesting Configuration Not Constrained to 100% or Specific Lifetime

INFO

Description: The vesting configuration instructions enforce basic sanity (non-zero percent and period counts) but do not guarantee that `vesting_percentage * max_vesting_periods` sums to exactly 100% of tokens:

```
  require!(vesting_percent > 0, PresaleError::InvalidVestingPercentage);
  require!(max_vesting_periods > 0,
PresaleError::InvalidVestingPercentage);
  global_state.vesting_percentage = vesting_percent;
  global_state.max_vesting_periods = max_vesting_periods;
```

`claim_vesting` correctly caps payouts so that `total_vested_tokens` never exceeds `tokens`, but if, for example, `vesting_percentage = 1500` and `max_vesting_periods = 10`, users would receive a smaller fraction per period and potentially never fully vest within the intended time horizon. Conversely, if the product exceeds 100%, all tokens will simply vest earlier than the nominal schedule suggests. This is primarily a UX and parameterization risk: without on-chain invariants enforcing a coherent schedule (e.g., `vesting_percentage * max_vesting_periods == 10000` basis points), misconfiguration could lead to confusing or unintended vesting behavior even though safety properties remain intact.

2. Methodology

A multi-layered approach was adopted. The following verification guidelines were applied:

1. **Rust & Anchor Code Quality**
   - Manual review of Rust/Anchor source code against the Rust API Guidelines, Solana program security recommendations, and Anchor best practices (no `unsafe`, `panic!`, or unchecked `unwrap`).
   - Verification that account structs, IDL, PDA seeds, and instruction handlers are consistent and minimal, with clear separation of concerns and no unused entrypoints.
   - Detection and removal of dead code, unused fields/variables, deprecated APIs, and redundant error variants that could conceal logic or expand attack surface.

2. **Logical & Functional Correctness**
   - Identification of logical bugs in instruction flow, including presale, vesting, and admin paths, with cross-checking against the intended specification and tokenomics.
   - Validation that account constraints, signer/owner checks, PDA derivations, and token/lamport invariants enforce correct initialization, updates, and invariants for `GlobalState`, `UserState`, vaults, and PDAs.
   - Confirmation that instruction return values, emitted events/logs, and state transitions behave as intended and that there is no hidden or malicious code enabling stealth drains or privilege escalation.

3. **Error-Handling & Resilience**
   - Assessment of error-handling via `Result`, custom error enums, and `require!` guards to ensure all failures are explicit, non-panicking, and leave no partial state updates.
   - Verification that all arithmetic uses checked operations where needed (overflow/underflow, division-by-zero), that timestamp/oracle bounds are validated, and that retries or repeated calls cannot bypass guard conditions.

- Examination of CPI failures, account-creation paths, and vesting/presale edge cases to confirm atomicity of operations and safe handling of rent-exempt balances, with no stranded lamports or tokens.

4. **Security, Authorization & Cryptography**
   - Analysis of cryptographic and oracle usage (Pyth feeds, PDA seed derivations, signature assumptions) for correct parameterization and to avoid custom or misused primitives.
   - Review of admin, PDA, and token-account authorization schemes, including ownership of vaults and upgrade authority configuration, to ensure strict access-control enforcement and least-privilege design.
   - Testing for Solana-specific vulnerabilities such as CPI-based reentrancy, numeric over/underflows, account-confusion and short-account attacks, and parameter tampering across instructions.
5. **Cross-Program & External Interactions**
   - Audit of all cross-program invocations (System, SPL Token, Associated Token, oracle clients) for correct account lists, ownership, seed usage, and absence of uninitialized or aliased account pointers.
   - Evaluation of interactions with external systems (oracles, governance, upgrade authority) to prevent race conditions, front-running on configuration changes, or inconsistent assumptions between programs.
   - Validation of program IDs, mint and oracle constants, address formats, feature flags, and import order to guarantee deterministic behavior across clusters and deployments.

6. **Operational Integrity & Resource Usage**
   - Confirmation that lamports and SPL tokens are correctly conserved, routed, and reserved, and that all state transitions are finalized without creating irrecoverable or dust balances.
   - Compute-unit and rent-cost checks to ensure predictable resource usage, absence of unbounded loops, and compatibility with Solana’s fee and rent model.
   - Verification that logs and events do not leak sensitive data, that all CPIs have their return values checked and errors propagated, and that there are no unchecked external calls or silent failure paths.

II. Verification checksums

Contract	Bytecode hash(SHA-256)
/src/constant.rs	8d820084aca0e855b80c2564fd609080b7cf602db1bea85f4fd546b632cfea75
/src/error.rs	ef54f8a887b8dc858ae2cad8641bb849583b2d669c6de25b7343d1a756a03d9e
/src/instructions/buy.rs	9303201cbb3ec6d90408fc0089f50f7889b372bec5ad9162461bde82af086823
/src/instructions/buy_with_stable_coin.rs	51e21951a53a2851e2223ff7070d9d8e7165ec5bf4befd004284b0feb1d67d88
/src/instructions/change_admin.rs	179e3d13423946f92930ef1571c1c52ef2937fc09d011919842f7741deeaa642
/src/instructions/claim_vesting.rs	15a8933b2ecf5b07e8a7fa1d08b762a78355f2cd5e62c82b6862a82f0c3f3ad5
/src/instructions/initialize.rs	013f8eb7efffc2ce52257736fc788c95c1ac87c1f6fb656e76a41001d50378d7
/src/instructions/mod.rs	f9902441165a2912db385d272b93e4883780832ccebc40d422f916b64d2155a7

/src/instructions/set_live.rs	00628c5dff15753eb00916ac0da267fea1f5764d56909f261dd8634a43d377e9
/src/instructions/set_price.rs	d44fd43379ae35a283c475ea9a40c43f6ac44c0af6adbd649abb75c7b284435e
/src/instructions/set_userdata.rs	1c640700d1e571f3aa1e0a99edfd1e190e4757dac10453ac88fc1b534f52abd1
/src/instructions/set_vault_address.rs	fcdaabc901ed8d6d4452c74a32332000c778e16a5a9f8413e0165a40867ebda1
/src/instructions/set_vesting_config.rs	05a7197e83291ddbc6d003d8c5ccf8abfed9aef884689278a076d2f6c2f9df14
/src/instructions/set_vestingtime.rs	f52488574617992d93adc42d7c810731a7a24860140263860dd334dd443f0423
/src/lib.rs	3733124922d25e6a7b399fbc7dbbd4b96b42309c2d05a1804b13baeb73d7df58
/src/state.rs	3eb40a1afcd9c2ad7b1852248af760fde1ac040a65c1f3b3d719c5d67a1d80ec
/src/util.rs	e30e57aa67c617d476e0d01d82c7cee1642eb63453fe502d6c232fc6a523c28f