SECURITY ASSESSMENT

for

Adrena

on

Apr 11, 2026

FIDESIUM

Table of Contents

Executive Summary **3**
Risk Issues **5**
Governance **8**
Remediation Roadmap **10**
Vulnerabilities **11**
Critical 11
High 12
Medium 16
Low 24
Info 35
Appendix **45**
Appendix A: Architecture Diagram 45
Appendix B: PDA Derivation Map 46
Appendix C: Token Flows 47
Appendix D: Onchain State 49
Appendix E: Authority Structure 50
Appendix F: Admin Powers 52
Appendix G: Permissionless Attack Surface 52
Appendix H: CPI Surface Analysis 53
Appendix I: Oracle Surface and Compound Attack Chains 55
Appendix J: Positive Observations 56
Appendix K: Dependency Analysis 64
Disclaimer **65**

Executive Summary

Report

TOTAL
Low risk
Apr 11, 2026

TOTAL
Low risk
Apr 05, 2026

TOTAL
Medium risk
Mar 18, 2026

Abstract

Autonom engaged Fidesium to assess the `adrena` program. This initial assessment was conducted between February 10, 2026 and March 18, 2026

Program Link:
https://github.com/AdrenaFoundation/adrena

Commit Hash:
`b7ab492090a66f096b593bafec00bfc197542570`

The follow-on assessment was conducted between March 31, 2026 and April 11, 2026

Commit Hash:
`94265a322a681a168d634da03c2e9b79dba1b4a3`

Key Findings

Primary risk areas: Oracle subsystem integrity (H2, H3, M2, M6) is the dominant concern, with three compound attack chains (Appendix I). AUM accounting gaps enable LP value extraction (H1). The absence of admin timelocks leaves no recovery path if governance is subverted (C1).

Significant risk: Missing slippage enforcement (M8) and permanent position DoS via zombie PDA position lockout (M4).

Bugs surfaced by automated methods:

- **Fuzzing:** borrow schedule dependence (L1), fresh pool AUM dipping below supply (I7), short borrow AUM asymmetry (I14), and stale funding rate crash (I12)
- **Property-based testing:** exponent sign error (M7)
- **Formal verification:** borrow and funding schedule dependence (L1) and LP quote/execution mismatch (I15)
- **Formal verification — existing findings confirmed:** Switchboard staleness laundering (H3), oracle silent downgrade (M6), PnL midpoint (M3), governance inflation (M1), Autonom ambiguous encoding (M5), off-peg fee settlement (L8), and short borrow AUM asymmetry (I14)

Scope

Source Files	168
SLOC	33,747
Instructions	125 public
State Structs	63

Codebase Profile

`instructions/`	24,027 lines
`state/`	10,912 lines
`lib/math/error/events`	1,716 lines
`adapters/`	577 lines

Report Suite

Document	Description
`Adrena.html`	Main audit report — all findings, risk scoring, remediation roadmap, and appendices A–K
`Adrena-v38-migration.html`	v38-specific behavioral differences, migration risks, and freeze window analysis
`Adrena-TestMatrix.html`	Detailed test matrix — PoC inventory, characterization tests, fuzzing architecture, and invariant catalog

Executive Summary

Issue Summary

Critical	High	Medium	Low	Info
~~1~~ - 0 Issue(s)	~~3~~ - 0 Issue(s)	~~8~~ - 0 Issue(s)	~~12~~ - 2 Issue(s)	~~18~~ - 1 Issue(s)

Engagement

Report Version	2.0 — Initial Delivery
Engagement Period	Feb 10 – Mar 18, 2026
Delivery Date	Mar 18, 2026
Review Period	Mar 31 – Apr 08, 2026
Delivery Date	Apr 11, 2026

Team

Lead Security Researcher	Abraham Polishchuk @skilurus
Security Researcher, Red Team Lead	Chadi Sebbar @DarOwn
Security Researcher	Gregory Bockenstette @bockus

Methodology

- Manual source code review
- System design analysis
- Fuzzing
- On-chain analysis
- Red-team exploit simulation
- Economic Modelling
- Formal verification
- Supply chain and dependency audit
- Compound attack chain analysis

Severity Definitions

Critical	The issue can cause large economic losses, large-scale data disorder or loss of control of authority management.
High	The issue puts users' sensitive information at risk or is likely to lead to catastrophic financial implications.
Medium	The issue puts a subset of users' sensitive information at risk, reputation damage or moderate financial impact.
Low	The risk is relatively small and could not be exploited on a recurring basis, or is low-impact to the client's business.
Informational	The issue does not pose an immediate risk but is relevant to security best practices or defense in depth.

Risk Issues

Active New Acknowledged Remediated Resolved

Vulnerability	Description	Risk	Probability	Status
C1: Missing Timelock on Admin Operations with single step transfer	Every privileged operation executes immediately, atomically, and with a single signature	Critical	Low	Resolved
H1: VFR Funding Obligations Invisible to AUM	AUM calculation ignores accrued Virtual Funding Rate (VFR) obligations.	High	Medium	Resolved
H2: Oracle guarantees degrade under staleness	When backup oracles are stale, `min_agree=1` pools trust the primary oracle as source of truth	High	Medium	Remediated
H3: Switchboard Oracle Staleness Laundering	Caller-Controlled `max_age_slots` and Timestamp Overwrite Enable Stale Price Acceptance Within a ~20-Second Window	High	Medium	Resolved
M1: Governance Inflation: Resolved Locked Stakes Recount on Sync	`sync_user_voting_power.rs` recalculates governance voting power by iterating all locked stakes, but only checks `amount > 0`. `is_resolved()` is never verified	Medium	High	Resolved
M2: Pixel-Perfect Liquidation via Stale Oracle Price	A signed oracle batch from a flash crash can be replayed to liquidate a recovered position. Up to the usage staleness check `staleness_seconds`, default 7s	Medium	Medium	Acknowledged
M3: PnL Uses Midpoint Instead of Protective Confidence Bands	`get_pnl_usd` comment states "Use High/Low price to protect the pool" but both Long and Short branches use `token_trade_price.price` (midpoint). Close and liquidation payouts are less conservative than intended when oracle confidence is nonzero.	Medium	Medium	Resolved
M4: Account revival: closed position PDAs can be zombified	When a position is fully closed, `close_position_long.rs` and `close_position_short.rs` drain lamports via `Cortex::transfer_sol_from_owned()` and zero struct fields via `*position = Position::default()`, but the 8-byte Anchor discriminator is not zeroed.	Medium	Medium	Resolved
M8: Missing Slippage Enforcement on Open Position	The `price` parameter in `open_position_long/short` and `increase_position_long/short` is documented as slippage protection but never compared to the execution price. Positions open at whatever the oracle reports, ignoring the user's limit. Close instructions enforce slippage via `MaxPriceSlippage` (v39), making this an asymmetry: users are protected on exit but not on entry.	Medium	Medium	Acknowledged
M5: Autonom Market-Opening Signatures Use Ambiguous Encoding	`autonom_market_opening_data.rs` constructs the signed message hash by concatenating two variable-length vectors without a length prefix or separator.	Medium	Low	Acknowledged
M6: Underweight Oracle Consensus	`min_agree = 3` Silently Behaves as `min_agree = 2`	Medium	Low	Resolved
M7: Incorrect exponent handling	`math.rs:124` uses `target_power += exponent1` where `checked_decimal_div` correctly uses `-=`, inflating results by `10^(2·exponent1)`	Medium	Info	Resolved
L1: Schedule-Dependent Interest and Funding Inflation via Ceiling Division Rounding	Splitting the same elapsed time into multiple `update_borrow_rate` calls inflates cumulative borrow interest because `checked_ceil_div` rounds up each sub-interval independently. Any permissionless operation that touches a custody triggers this rounding, allowing an attacker to grief borrowers at negligible cost.	Low	Medium	Acknowledged
L2: Hardcoded RPC Authentication Tokens in Source	Three CLI files contain hardcoded RPCPool authentication tokens embedded in URLs	Low	Medium	Remediated

Risk Issues

Active New Acknowledged Remediated Resolved

Vulnerability	Description	Risk	Probability	Status
L3: DOS: Residual Unchecked Oracle Math	`oracle.rs` retains plain arithmetic in `OraclePrice::low()` and `OraclePrice::high()`	Low	Low	Resolved
L4: Griefing: Pool-Wide Write-Lock	Permissionless `distribute_fees` can be called as a no-op to write-lock the pool, all staking accounts, and the oracle PDA for an entire block, blocking every concurrent position and staking operation	Low	Low	Resolved
L5: Griefing: Liquidation Capture	Permissionless `resolve_position_borrow_fees` can be frontrun and defer collateral increases/position closes	Low	Low	Acknowledged
L6: Update-Time Lock Traps User After `add_collateral`	Calling `add_collateral` resets `update_time`, locking the user out of `close_position` for `min_position_update_time_before_close_seconds` (default 120s). During this window the price can continue moving against the user.	Low	Low	Acknowledged
L7: Governance Lacks Participation Quorum and Execution Timelock	The AdrenaDAO governance (`HgeoVqTTMQ9K5GZAUpPKaz5PS8Rn55yR5e5SwmB3DbKB`) is configured with a `YesVotePercentage(45%)` threshold, which requires 45% of *deposited* votes to approve — not 45% of total token supply. There is no minimum participation quorum, no execution timelock (`transactions_hold_up_time = 0`), and no council or veto body.	Low	Low	Acknowledged
L8: Fee Distribution: Off-Peg Stablecoin Creates Phantom Pool Value	`distribute_fees.rs:257-261` sets token transfer amounts directly from USD-denominated fee debt buckets with no price conversion.	Low	Low	Resolved
L9: Feeless Swap Skips Borrow Rate Update	Internal (feeless) swaps modify `custody.assets.owned` but return early before calling `update_borrow_rate` on either custody. The borrow rate remains based on pre-swap utilization until the next instruction that touches the affected custody.	Low	Low	Resolved
L10: Griefing: Deprecated Permissionless IDL	`genesis_stake_patch.rs` appears as callable in IDL even though it is deprecated	Low	Info	Resolved
L11: LP Sandwich: No Structural Hold Period	LP tokens can be minted and redeemed in consecutive slots with no time-lock enforced	Low	Info	Resolved
L12: Wrapping Overflow in Financial Calculation	`pool.rs` uses wrapping computation for numbers with financial impact	Low	Info	Acknowledged
I1: Deprecated Instruction: Permissionless Account Corruption	`migrate_pool_v37_to_v38.rs` does not validate PDA seed or discriminator	Info	Info	Resolved
I2: Deprecated Callable IDL Instruction	`patch_custody_locked_amount.rs` appears as callable in IDL even though it is deprecated	Info	Info	Resolved
I3: Shared PDA Namespace Between accounts	Both `add_custody` and `add_synthetic_custody` derive PDAs from `["custody", pool, seed]` with an arbitrary `[u8; 32]` seed.	Info	Info	Resolved
I4: Unchecked Subtraction	`Position::subtract` uses `-=`, which will revert with a generic panic due to `overflow-checks = true`	Info	Info	Acknowledged
I5: Referrer Fee Stripping: Optional Account Omission	Referrer fee accrual in `pool.rs:accrue_fee_debt` depends on the caller-supplied `has_referrer` boolean, which is derived from whether optional `user_profile` and `referrer_profile` accounts are passed.	Info	Info	Acknowledged

Risk Issues

Active New Acknowledged Remediated Resolved

Vulnerability	Description	Risk	Probability	Status
I6: Permissionless Migration: Nickname Front-Run and Snipe	`migrate_user_profile_from_v1_to_v2` accepts any `caller: Signer` with no owner gate, allowing a third party to consume the one-shot V1-V2 migration for any legacy profile.	Info	Info	Acknowledged
I7: Fresh Pool AUM can transiently dip below supply	`add_liquidity` accrues fee_debt. On a fresh pool this means AUM briefly dips below supply.	Info	Info	Acknowledged
I8: Authority Convergence: BPF Upgrade Authority and Cortex Admin	Both the BPF Loader upgrade authority and the on-chain `Cortex.admin` are set to the same address: the DAO native treasury `7VzEXYvGmLg3tdVuFuGFQdr7GP5tutTUt8EcTGHvG8Ev`.	Info	Info	Acknowledged
I9: Low Proposal Creation Threshold Enables Governance Spam	The governance configuration sets `min_community_weight_to_create_proposal` to 1,000,000 tokens, which represents 0.058% of the total 1.72B token supply. This is a very low barrier for proposal creation.	Info	Info	Acknowledged
I10: Dead Code: Partially Unused SPL Governance CPI Adapter	The file `adapters/spl_governance_adapter.rs` defines 8 CPI wrapper functions for the SPL Governance program. Of these, 3 are actively used	Info	Info	Acknowledged
I11: No runtime invariant between dual OI tracking systems	Funding rates and AUM/PnL compute distinct Open Interest	Info	Info	Acknowledged
I12: Stale Funding Rate After Config Downgrade	`set_custody_virtual_funding` overwrites `custody.virtual_funding` config parameters but does not recalculate or clamp the live `current_rate_long_to_short` stored in `custody.virtual_funding_state`.	Info	Info	Acknowledged
I13: Core Dependencies outdated	The Solana SDK and Anchor are out of date with latest versions.	Info	Info	Resolved
I14: Short Borrow Interest AUM Asymmetry	The AUM calculation treats long and short borrow interest asymmetrically. Long unrealized interest is subtracted from AUM continuously (`pool_value_sub_usd += long_pnl.borrow_fee_usd`, pool.rs), but short unrealized interest is not.	Info	Info	Acknowledged
I15: Liquidity Quote View Overstates LP Tokens	The `get_add_liquidity_amount_and_fee` view instruction returns more LP tokens than `add_liquidity` actually mints on the same pool state. The view divides by `pool.aum_usd` (cached), while execution divides by `(pool_amount_usd + lp_fee_usd)`	Info	Info	Acknowledged
I16: No Insurance Fund or Bad Debt Tracking	When a liquidated position's loss exceeds its collateral and accrued fees, the protocol silently absorbs the deficit into the pool. The `get_exit_position_numbers` function in `pool.rs` computes two deficit values: `deficit_fee_usd` (uncollected fees) and `deficit_pool_usd`. These are never persisted on-chain.	Info	Info	Acknowledged
I17: No Verified Build Pipeline or Source Integrity CI	No deterministic build pipeline exists in the repository. Fidesium independently verified the current deployment as a bit-exact match against `release/37_4` (commit `5242fcdb`), but future deployments lack this guarantee. No Trojan Source CI check exists (source is currently clean).	Info	Info	Resolved
I18: Fee Debt Accrued at Mid Price, AUM Assets Valued at Low Price	There is a fee debt price inconsistency between `custody.assets.owned` and debt	Info	Info	Resolved

Governance

Governance Trust Model

The protocol's user-facing operations are trust-minimized via PDA-derived accounts, deterministic fee calculations, and oracle-priced exits. All configuration is controlled by a single `cortex.admin` pubkey with no program-enforced constraints on what it can do or how quickly it can act. This section documents the complete admin attack surface and the trust assumptions users are making.

Admin Control Surface

52+ admin-gated instructions exist across pool, custody, staking, and cortex configuration. The following are the highest-impact levers:

Instruction	What it controls	Blast radius
`set_admin`	Transfer admin authority to any pubkey	Permanent takeover. No timelock, no two-step, no governance vote (C1)
`set_pool_fee_config`	Fee shares, manager recipient, pool type, oracle provider	Redirect up to 100% of fees to attacker wallet. Validation only checks sum = 10,000 BPS
`set_all_pools_fee_shares`	Fee shares across all pools atomically	Single transaction reconfigures every pool
`set_protocol_fee_recipient`	Protocol fee destination wallet	Redirect DAO revenue to attacker
`set_custody_config`	Fees, pricing, borrow rates, oracle, leverage limits	Set `close_position` fee to 100% BPS. Lower `max_leverage` to force-liquidate all positions
`set_pool_liquidity_state`	Toggle Active/Idle	Freeze LP redemptions. No global pause exists; must disable per-pool
Program upgrade (off-chain)	Replace entire program binary	Total control. All on-chain state persists, new code has unrestricted access

Fee Parameter Bounds

Custody fee parameters are validated by `Fees::validate()` (custody.rs:315-325), which checks each fee field is `<= BPS_POWER` (10,000). This means **100% fees are valid**:

Parameter	Bound	Consequence at max
`fees.close_position`	≤ 10,000 BPS (100%)	Every close costs the entire position value in fees
`fees.liquidation`	≤ 10,000 BPS (100%)	Every liquidation costs the entire position value in fees
`pricing.max_leverage`	≥ `max_initial_leverage` ≥ `MIN_INITIAL_LEVERAGE` (1.1x)	Lowering to 1.1x makes every existing leveraged position instantly liquidatable
`borrow_rate.max_hourly_borrow_interest_rate`	≤ `RATE_POWER` (100%/hr)	100% hourly borrow rate drains borrowers within one hour

No economic bounds (e.g., "close fee cannot exceed 5%") are enforced. The validation is purely structural (does it fit in the field?).

Maximum Damage Chain

A compromised admin key can drain the protocol in approximately 5 transactions within a single Solana block:

Governance

Governance Trust Model

Governance Enforcement Gap

The protocol has SPL Governance v4 infrastructure configured (`cortex.governance_program`, `cortex.governance_realm`) with ADX-staking-weighted voting power. **No admin instruction checks governance state**. The `set_admin` handler (set_admin.rs:31-33) performs `cortex.admin = params.new_admin`.

Governance enforcement depends entirely on the admin key being a Realms proposal executor PDA. If it is, then the Realms program enforces proposal flow. If it is not (e.g., a team wallet, a compromised key, or an admin transferred via `set_admin` outside governance), the DAO has no recourse within the Adrena program. L7 documents the governance configuration weaknesses.

The admin cannot silently siphon tokens. Any attack produces visible on-chain fee/liquidation events.

User Trust Assumptions

Users of Adrena are implicitly trusting that:

1. The admin key is controlled by a properly governed multisig with adequate delay
2. The program upgrade authority is held by the same governance body
3. Fee parameters will not be changed to confiscatory levels between position open and close
4. Leverage limits will not be lowered to force-liquidate existing positions

These assumptions are standard for managed DeFi protocols. See C1 for the primary recommendation (timelock + two-step transfer) and L7 for governance configuration hardening.

Remediation Roadmap

Priority	Finding	Action	Rationale	Effort
Immediate	H3: Switchboard Staleness Laundering	Clamp timestamp at storage time: `stored_price.timestamp = new_price.timestamp.min(current_time)`	Force multiplier for three compound attack chains (Appendix I). Defuses H2, M2, and M6 exploitation paths simultaneously.	Small: single line change in `oracle.rs`
Immediate	C1: Missing Admin Timelock	Two-step admin transfer (propose + accept after delay). Add configurable timelock on parameter-mutation instructions (48h oracle/custody, 24h fee shares).	Rated Critical. Admin is already behind DAO governance, which lowers probability, but the single-step transfer pattern has no recovery path if governance is subverted.	Moderate: new state fields, pending-admin PDA, timelock queue
Short-term	H1: VFR Funding Obligations Invisible to AUM	Include unrealized funding obligations in `get_assets_under_management_usd`. Surface `unrealized_funding_paid/received_usd` from `get_collective_position()` instead of zeroing via `..Position::default()`.	LP redemption at inflated AUM socializes funding costs to remaining LPs. Attacker-as-LP variant demonstrated pool drain in PoC.	Moderate: design decision on collective position aggregation, AUM formula change, LP pricing impact
Short-term	H2: Oracle Consensus Degradation	Require at least one backup oracle for liquidation operations under `min_agree=1`. Add per-pool `last_consensus_slot`. Pause liquidations after N slots of single-oracle operation as a circuit breaker.	Compounds with H3 (Appendix I, Chain 1). Autonom pools operate in `min_agree=1` by default.	Moderate: new pool state field, asymmetric liquidation gate, circuit breaker logic
Short-term	M1: Governance Resolved Stake Inflation	Add `!is_resolved()` gate in `sync_user_voting_power` locked stake iteration. Consider migration sweep to correct existing phantom voting power.	High probability: permissionlessly exploitable. Any account can call sync to re-inflate resolved stakes. Governance outcomes at risk.	Small: one conditional added to loop filter. Migration sweep is optional but recommended.
Short-term	M4: Zombie PDA Position DoS	Add Anchor `close` attribute to `close_position_long/short` to fully zero accounts on close, preventing account revival. Add admin force-close instruction for existing zombie PDAs.	Working PoC demonstrates permanent position slot lockout at ~0.002 SOL per slot. No recovery path exists without a program upgrade.	Small: replace manual lamport drain with Anchor close. Moderate for admin recovery instruction.
Short-term	M8: Missing Slippage Enforcement on Open	Add the same `MaxPriceSlippage` check from close to open/increase instructions. Close already enforces slippage (v39); open does not.	Affects every position entry. During congestion, users are exposed to adverse execution with up to 83% collateral loss. Proven on v39 production binary via surfpool.	Small: additional parameter and price check on each position instruction.
Medium-term	M2–M3, M5–M7, L1–L12	Address remaining Medium and Low findings per individual writeups. Prioritize M5 (Autonom ambiguous encoding) next, as it has a working PoC with demonstrated economic impact.	Defensive hardening. No compound escalation paths identified among these findings in isolation.	Variable. See individual findings

Vulnerabilities Critical

Missing Timelock on Admin Operations with single step transfer

Vulnerability severity:	Critical	Vulnerability probability:	Low	Status:	Resolved

Admin transfer now uses a two-step process with a fixed 48-hour delay. The old single-step admin transfer path is no longer usable, and the current admin can also cancel a pending transfer. This removes the immediate admin-takeover risk identified in the original finding.

Every privileged operation executes immediately, atomically, and with a single signature

A stolen or compromised admin key would grant immediate unrestricted protocol control

In the worst case, a three transaction takeover of the protocol, followed by freeze, and then fee drain would lead to total protocol failure.

Additionally, users could be trapped in their positions, accruing borrow fees with no exit path, while the attacker redirects fee distribution via `set_all_pools_fee_shares`, causing all fees accruing against trapped positions to flow to the attacker's wallet rather than legitimate liquidity providers.

Recommendations:

Implement a two-step admin transfer (propose + accept after a delay), and add a configurable timelock (suggested: 48 h for oracle/custody config, 24 h for fee shares) on all parameter-mutation instructions. Alternatively, transfer admin authority to an on-chain governance multisig before mainnet launch.

Vulnerabilities High

Oracle guarantees degrade under staleness

Vulnerability severity:	High	Vulnerability probability:	Medium	Status:	Remediated

Remediated at commit `103093b8` by introducing configurable asymmetric liquidation, circuit breakers, and falling back to this when `min_agree = 1`. However, `has_any_fresh_backup_price` checks whether any non-primary price for any asset is fresh

```
for price in self.prices.iter() {
      if price.feed_id >= primary_min && price.feed_id <= primary_max {
          continue;
      }
      if price.check_price_validity(current_time, cb_seconds).is_ok() {
          return Ok(true);
      }
  }
```

This means a compromised primary feed can be used for liquidations, so long as any price for any asset is being published by a secondary, which is essentially always true (USDC). Additionally, the checks live in `liquidate_long`, `liquidate_short` directly. This leaves other instructions vulnerable when backups are stale. Autonom pools default to both defense layers disabled.

The postaudit code adds a stricter three-oracle agreement mode. However, the vulnerable single-agreement path remains in place for the affected configuration: if backup oracles are stale or unavailable, the system can still rely on a single oracle price. Based on the current code and verification results, this finding remains open.

When backup oracles are stale, `min_agree=1` pools trust the primary oracle as source of truth

Autonom pools use `min_agree = 1`

When secondary/tertiary oracles fall outside staleness window, `oracle.rs#get_selected_price_by_feed_id_or_name` defaults to trusting the primary oracle by default.

```
if pool_cfg.min_agree == 1 {
    if let Some(p) = primary_price {
        if let (Some(s), Some(t)) = (secondary_price, tertiary_price) {
            if within_threshold(p.price, s.price, pool_cfg.price_diff_threshold_bps)?
                || within_threshold(p.price, t.price, pool_cfg.price_diff_threshold_bps)?
            {
                return Ok(p);
            }
            return Ok(s);
        }
        return Ok(p);
    }
}
```

An attacker could monitor chain for congestion, and wait for the backup oracle cranks to fall outside the 7s window. Alternatively a malicious/colluding crank operator could deliberately delay their own updates.

This is compounded by `oracle.rs#verify_and_update_pricesstored_price.timestamp = new_price.timestamp;`

Since `MAX_FUTURE_DRIFT` is 60s, this effectively provides a 67s window.

Vulnerabilities High

CONTINUED: Oracle guarantees degrade under staleness

Recommendations:

Given that `min_agree = 1` is a valid protocol setting, there are a handful of remediation options

- Apply asymmetric rules, and require at least one backup oracle for liquidation operation.
- Clamp oracle freshness at storage time: `stored_price.timestamp = new_price.timestamp.min(current_time);`
- Default to TWAP price feeds
- Apply a circuit breaker - track a per pool `last_consensus_slot`. If the oracle has been operating in single-oracle mode for more than N slots, pause liquidations entirely.

Switchboard Oracle Staleness Laundering

Vulnerability severity:	High	Vulnerability probability:	Medium	Status:	Resolved

Resolved at commit `61f867fa` and further refined in commits `b95afdfe`, and `042fac5a` by clamping timestamps at storage to `current_time`, and rejecting `if age_slots > params.max_age_slots {`

Caller-Controlled `max_age_slots` and Timestamp Overwrite Enable Stale Price Acceptance Within a ~20-Second Window.

NOTA BENE: As demonstrated in Appendix I, this compounds with the wider Oracle attack surface, escalating from Medium to High due to cross vulnerability compound attack risk.

`SwitchboardUpdateParams::max_age_slots` is a caller-supplied parameter with no on-chain cap. The `orac1e` program enforces a hardcoded `max_age=50` slots (~20s), but within that window Adrena's timestamp overwrite makes stale data appear fresh. Adrena's own slot-based staleness check is redundant since `max_age_slots` is caller-controlled:

```
let quote_age_slots = current_slot.saturating_sub(parsed_quote.slot);
if quote_age_slots > params.max_age_slots {
    return Err(AdrenaError::SwitchboardQuoteTooStale.into());
...
...
if stored_price.timestamp < current_time {
    stored_price.timestamp = current_time;  // ← launders the staleness
    stored_price.price = new_price;
    stored_price.confidence = get_confidence_from_price(new_price, *required_feed_id)?;
}
```

Downstream, `check_price_staleness_with` compares against this laundered timestamp, so up to ~20-second-old data passes as fresh. `update_oracle` is fully permissionless. Adrena's instruction sysvar check validates only the shape of a preceding Switchboard instruction (program ID, queue, writable quote) but does not verify content or Ed25519 precompile presence independently.

Attack scenario: An attacker captures a legitimately signed Switchboard quote at price X, then waits up to ~20 seconds for the market to move to price Y. They replays the old quote via `update_oracle`. The attacker opens a position at the stale price X. When an honest keeper updates the oracle to the real price Y, the attacker closes for the delta.

With `min_agree = 1` pools, the primary oracle is trusted as sole source of truth when backups are stale (see H2). With `min_agree = 2`, an attacker controlling two Switchboard quote accounts (secondary + tertiary) can inject two stale prices that agree with each other, satisfying consensus with fabricated agreement.

Vulnerabilities High

CONTINUED: Switchboard Oracle Staleness Laundering

orac1e Program Verification (Mainnet Simulation): The Switchboard quote program (`orac1eFjzWL5R3RbbdMV68K9H6TaCVVcL6LjvQQWAbz`) is closed-source and upgradeable. It currently enforces Ed25519 precompile and signed data freshness with hardcoded `max_age=50` slots (~20s, `verified_update.rs:42`). These defenses are real but external to Adrena. Switchboard is upgradeable.

Recommendations:

- Verify Ed25519 precompile is present

```
fn verify_ed25519_precompile_present(
    instructions_sysvar: &AccountInfo,
      current_instruction_index: usize,
  ) -> Result<()> {
      for i in 0..current_instruction_index {
          let ix = load_instruction_at_checked(i, instructions_sysvar)
              .map_err(|_| AdrenaError::SwitchboardInvalidQuoteInstruction)?;

          if ix.program_id == ed25519_program::id() {
              return Ok(());
          }
      }

      msg!("No Ed25519SigVerify precompile instruction found before update_oracle");
      Err(AdrenaError::SwitchboardInvalidQuoteInstruction.into())
}
```

- Verify the Ed25519 instruction references the same quote account's data by parsing the Ed25519 instruction layout and matching the signed message to the quote account content
- Hardcode `max_age_slots` as a protocol constant or store it in the pool/oracle config under admin control:

```
const MAX_SWITCHBOARD_AGE_SLOTS: u64 = 10;
```

- Implement a per-provider cooldown or rate limit on `update_oracle` to prevent rapid price manipulation within the monotonicity guard's one-second window

Vulnerabilities High

VFR Funding Obligations Invisible to AUM

Vulnerability severity:	High	Vulnerability probability:	Medium	Status:	Resolved

Resolved in full. Two new fields cumulative_funding_paid_usd and cumulative_funding_received_usd were added to PositionsAccounting (repurposing padding bytes), and get_collective_position() now populates unrealized_funding_paid_usd / unrealized_funding_received_usd from these aggregates instead of zeroing them via ..Position::default(). AUM now reflects VFR funding obligations.

The residual inline settlement issue (where close_position_long/short updated position.unrealized_funding_paid/received_usd += directly, bypassing aggregate tracking) has also been resolved: a new custody.settle_funding_on_position() method atomically updates both the position and the custody-level aggregates, and all 10 instruction paths (close_position_long/short, liquidate_long/short, increase_position_long/short, add/remove_collateral_long/short) now call it. No inline += settlement remains outside of this method. Position removal paths correctly decrement the aggregates via saturating_sub.

AUM calculation ignores accrued Virtual Funding Rate (VFR) obligations.

LP token redemption is priced against this mispriced AUM, allowing LPs to exit at inflated value while remaining LPs absorb the funding outflow when positions close.

The pool acts as economic counterparty for funding payments, but `get_collective_position()` zeroes all funding fields via `..Position::default()`

```
Ok(Position {
    side: side.into(),
    price: /* weighted avg */,
    size_usd: accounting.size_usd,
    ..Position::default()  // zeroes unrealized_funding_paid/received_usd
})
```

When `get_assets_under_management_usd()` feeds this into `get_pnl_usd()`, the funding component evaluates to zero. AUM does not reflect funding obligations owed to minority-side positions.

LP redemption has zero cooldown, zero lock period, and no daily limit. The only friction is a 0.10% fee. No cross-wallet position checks exist, so an attacker can hold offsetting positions via separate wallets.

The attacker can create the imbalance, harvest the funding, and redeem LP tokens before the obligation materializes. This is repeatable with no cooldown.

Recommendations:

Track aggregate funding at the custody accounting level. Add `unrealized_funding_paid_usd` and `unrealized_funding_received_usd` to `PositionsAccounting`, update on every position open/close/settle, and populate in `get_collective_position()` so AUM reflects funding obligations.

```
// PositionsAccounting (new fields):
pub unrealized_funding_paid_usd: u64,
pub unrealized_funding_received_usd: u64,

// get_collective_position() — replace ..Position::default():
unrealized_funding_paid_usd: accounting.unrealized_funding_paid_usd,
unrealized_funding_received_usd: accounting.unrealized_funding_received_usd,
```

Additionally, consider an LP withdrawal fee tier that increases during periods of high OI imbalance, reducing the profitability of timing-based LP extraction.

Implementing an LP hold period (see L11) would also partially mitigate this finding by preventing instant redemption against mispriced AUM, adding friction to the redemption step of the attack.

Vulnerabilities Medium

Governance Inflation: Resolved Locked Stakes Recount on Sync

Vulnerability severity:	Medium	Vulnerability probability:	High	Status:	Resolved
Resolved at commit `61f867fa` by adding `!locked_stake.is_resolved()` to the gate.					

`sync_user_voting_power.rs` recalculates governance voting power by iterating all locked stakes, but only checks `amount > 0`. `is_resolved()` is never verified

```
for locked_stake in user_staking.locked_stakes.iter() {
    if locked_stake.amount > 0 {
        real_voting_power += checked_as_u64(
            locked_stake.vote_multiplier as u128
                * locked_stake.amount as u128
                / Cortex::BPS_POWER,
        )?;
    }
}
```

`finalize_locked_stake.rs` correctly revokes voting power via `remove_governing_power` CPI, then sets `locked_stake.resolved = true`. It does not zero `amount`, because `remove_locked_stake` needs it to return tokens to the user.

After finalization, any account can call the permissionless sync_user_voting_power instruction to force a recount. Because resolved stakes still have amount > 0, the sync re-adds the full voting power that finalize_locked_stake just revoked. The attacker retains their tokens and their governance voting power — effectively double-spending their stake. With all MAX_LOCKED_STAKE_COUNT = 32 slots finalized, the attacker recovers the entirety of their staked voting power while holding zero locked tokens.

This is particularly concerning given its interaction surface with `C1: Missing Timelock on Admin Operations with single step transfer`

Total Governance Capture (50%+1) occurs at 21.6% of circulating supply locked for 540 days (4x multiplier). At current market rates ($0.0009296 per ADX on March 18, 2026), this would cost just north of 200k USD.

That said, this does allow existing stakers to exit their position but keep their voting power.

Recommendations:

Add an `is_resolved()` guard to the voting power loop in `sync_user_voting_power`:

```
for locked_stake in user_staking.locked_stakes.iter() {
  if locked_stake.amount > 0 && !locked_stake.is_resolved() {
      real_voting_power += checked_as_u64(
          locked_stake.vote_multiplier as u128
              * locked_stake.amount as u128
              / Cortex::BPS_POWER,
      )?;
  }
}
```

Vulnerabilities Medium

Account revival: closed position PDAs can be zombified

Vulnerability severity:	Medium	Vulnerability probability:	Medium	Status:	Resolved

Resolved at commit `61f867fa` by adding

```
if !is_partial_close {
    let mut data = ctx.accounts.position.as_ref().try_borrow_mut_data()?;
    data[..8].fill(0);
}
```

Since Anchor is a 0.31.1, pre funded accounts are explicitly handled at `init` time and DoS is no longer possible

When a position is fully closed, `close_position_long.rs` and `close_position_short.rs` drain lamports via `Cortex::transfer_sol_from_owned()` and zero struct fields via `*position = Position::default()`, but the 8-byte Anchor discriminator is not zeroed.

The discriminator lives outside the `DerefMut` target of `AccountLoader<Position>`, so `Position::default()` only writes to bytes 8..N, leaving bytes 0..8 intact.

```
Cortex::transfer_sol_from_owned(
    ctx.accounts.position.to_account_info(),
    ctx.accounts.owner.to_account_info(),
    ctx.accounts.position.get_lamports(),
)?;

...
*position = Position::default();
```

Accounts drained to 0 lamports are garbage-collected at end of transaction. If an attacker sends lamports back to the PDA within the same transaction via a bundled `system_program::transfer` instruction, the account revives with a valid discriminator but all-zero struct fields.

`close_position_long/short` can be called **permissionlessly** when stop-loss or take-profit triggers are reached

```
if !ctx.accounts.caller.key.eq(ctx.accounts.owner.key) {
    if !is_affected_by_stock_event {
        require!(
            (position.take_profit_is_set()
                && position.take_profit_reached(token_trade_price.price))
                || (position.stop_loss_is_set()
                    && position.stop_loss_reached(token_trade_price.price)),
            AdrenaError::InvalidPositionState
        );
```

The zombie PDA creates a permanent denial-of-service on the victim's position slot. Recovery is impossible

The attack costs the rent exempt minimum of ~0.002 SOL per position slot. To fully DoS a user across 3 custodies × 2 sides = ~0.012 SOL.

Direct profit extraction is not possible since the bump mismatch blocks all instructions from operating on the zombie. The attack enables competitive griefing and selective liquidity removal

Vulnerabilities Medium

CONTINUED: Account revival: closed position PDAs can be zombified

Recommendations:

Conditionally call Anchor's `close()` to fully reassign the account to the system program, and reallocate data to 0 bytes, rendering it reinitializable.

```
ctx.accounts.position.close(ctx.accounts.owner.to_account_info())?;
```

Pixel-Perfect Liquidation via Stale Oracle Price

Vulnerability severity:	Medium	Vulnerability probability:	Medium	Status:	Acknowledged

Acknowledged by adrena team with comment: Accepted. If ChaosLabs attested a price at which a position was underwater, that liquidation was legitimate at that moment. The monotonic timestamp guard prevents replay once a newer batch is stored, and the 7s ceiling bounds execution latency. This matches standard perp semantics — crossing the liquidation threshold at any attested price point is a valid trigger regardless of subsequent recovery.

A signed oracle batch from a flash crash can be replayed to liquidate a recovered position. Up to the usage staleness check `staleness_seconds`, default 7s

There are two separate staleness gates that interact:

```
if self.timestamp < current_time - MAX_TIMESTAMP_AGE {
    return Err(AdrenaError::StaleOraclePrice.into());
}
...
...
pub fn check_price_staleness_with(
    &self,
    current_time: i64,
    staleness_seconds: i64,
) -> Result<()> {
    require!(
        self.timestamp + staleness_seconds >= current_time,
        AdrenaError::StaleOraclePrice
    );
    Ok(())
}
```

LPs profit from the wrongful liquidation as the pool absorbs the position's collateral at a stale price.

Since the liquidation fee goes to the pool, the attacker would need to hold a substantial LP position, and monitor for the remaining conditions (flash crash, network congestion, positions near liquidation threshold)

Vulnerabilities Medium

CONTINUED: Pixel-Perfect Liquidation via Stale Oracle Price

Recommendations:

Tighten `MAX_TIMESTAMP_AGE` to near `staleness_seconds` (e.g. 10s) so that ingestion rejects batches that would pass the usage check but reflect outdated market conditions. Additionally, add a monotonic timestamp guard for liquidations: reject a batch if the oracle already holds a newer price.

```
   pub const LIQUIDATION_MAX_TIMESTAMP_AGE: i64 = 10;
.....
   if new_price.timestamp <= existing_price.timestamp {
       return Err(AdrenaError::StaleOraclePrice.into());
   }
```

Note: This finding compounds with H3 (Switchboard Staleness Laundering), which extends the effective replay window from 7s to ~20s by laundering the stored timestamp. Fixing H3's timestamp overwrite (see Appendix I: Oracle Surface and Compound Attack Chains, Chain 2) neutralizes the most severe exploitation path for this finding.

PnL Uses Midpoint Instead of Protective Confidence Bands

Vulnerability severity:	Medium	Vulnerability probability:	Medium	Status:	Resolved
Resolved at commit `61f867fa` by adding `conservative_pricing` parameter					

`get_pnl_usd` in `pool.rs` computes exit price for both Long and Short positions using the oracle midpoint, despite a comment documenting the intent to use protective confidence bands:

```
// Use High/Low price to protect the pool
let exit_price = match Side::try_from(position.side)? {
    Side::Long => token_trade_price.price,
    Side::Short => token_trade_price.price,
    Side::None => return Err(AdrenaError::InvalidPositionState.into()),
};
```

`OraclePrice` already exposes `.low()` and `.high()` methods that subtract/add confidence, and the AUM calculation paths use them correctly:

```
// In get_pool_info_snapshot_pda.rs — AUM uses protective pricing
let long_pnl = pool.get_pnl_usd(
    &long_collective_position,
    &token_trade_price,
    &token_price.high(),  // confidence-adjusted
    ...
);
```

Vulnerabilities Medium

CONTINUED: PnL Uses Midpoint Instead of Protective Confidence Bands

Close position and liquidation callers pass the raw oracle price without adjustment. During volatile periods when oracle confidence widens, this results in exit prices that are more favorable to traders and less protective of the pool than intended.

For longs, the pool-protective exit price should be `token_trade_price.low()` (lower price = less profit for trader). For shorts, it should be `token_trade_price.high()` (higher price = less profit for trader).

Recommendations:

Apply confidence bands in the exit price match:

```
let exit_price = match Side::try_from(position.side)? {
    Side::Long => token_trade_price.low().price,
    Side::Short => token_trade_price.high().price,
    Side::None => return Err(AdrenaError::InvalidPositionState.into()),
};
```

Autonom Market-Opening Signatures Use Ambiguous Encoding

Vulnerability severity:	Medium	Vulnerability probability:	Low	Status:	Acknowledged

Acknowledged by Adrena team with

1. The Autonom signer controls both vectors and signs the intended split.
2. `autonom_market_opening.rs` validates every feed ID against real synthetic custodies — a misinterpreted split would fail validation.
3. Feed sets are deterministic per market configuration, not user-supplied.
4. Fixing requires coordinating both the on-chain program and the Autonom backend signer simultaneously — planned for a future coordinated release.
5. A dedicated Autonom endpoint (`GET /hours/batch/adrena`) is being built to sign market opening data using the exact on-chain byte layout (u8 feed IDs + i64 LE timestamps + keccak256). A middleware service fetches the signed payload and submits it on-chain via the `autonom_market_opening` instruction. The encoding ambiguity is contained within the trusted signer boundary.

Vulnerabilities Medium

CONTINUED: Autonom Market-Opening Signatures Use Ambiguous Encoding

`autonom_market_opening_data.rs` constructs the signed message hash by concatenating two variable-length vectors without a length prefix or separator.

```
pub fn build_message_hash(&self) -> Result<[u8; 32]> {
    let mut msg = vec![];
    msg.extend_from_slice(&self.market_open_timestamp.to_le_bytes());
    msg.extend_from_slice(&self.market_close_timestamp.to_le_bytes());

    self.feeds.iter().for_each(|feed_id| msg.push(*feed_id));
    self.market_close_affected_feeds
        .iter()
        .for_each(|feed_id| msg.push(*feed_id));

    Ok(hashv(&[msg.as_slice()]).to_bytes())
}
```

The byte boundary between `feeds` and `market_close_affected_feeds` is not committed to the hash. Two logically different payloads (for example `feeds=[40], close_affected=[40]` and `feeds=[40, 40], close_affected=[]`) produce the identical hash because the concatenated bytes are `[40, 40]` in both cases. One signature therefore authenticates multiple distinct decompositions.

The handler's validation in `autonom_market_opening.rs` checks set containment between `feeds` and the pool's synthetic custody feed IDs, but does not reject duplicates

An attacker who observes a legitimately-signed `autonom_market_opening` transaction on-chain can slide bytes between the two vectors and re-submit a transaction that passes both signature verification and the handler's feed validation, but writes different `market_close_affected_feeds` into pool state.

Two attack scenarios:

1. **Liquidation shield removal:** A stock undergoes a 2:1 split overnight. The oracle signer marks the feed as split-affected via `feeds=[40], close_affected=[40]`, which blocks liquidation (`liquidate_long.rs`) and forces affected positions to close at the pre-split price. The attacker re-submits with `feeds=[40, 40], close_affected=[]` — same hash, same signature, duplicate passes the set-containment check. The pool's `market_close_affected_feeds` is now empty. The post-split oracle reports the halved price, making long positions appear deeply underwater. The attacker calls `liquidate_long` and extracts the liquidation reward from positions the protocol intended to protect. **Integration test confirmed: liquidation blocked before the attack, succeeds after.**
2. **Phantom affected-feed injection:** If the signer ever signs a payload with a duplicate in `feeds` e.g. `feeds=[40, 40], close_affected=[]`, an attacker can reinterpret it as `feeds=[40], close_affected=[40]`, falsely marking a feed as split-affected. This blocks `increase_position_long` (`increase_position_long.rs`, error `MarketStockSpecialEvent`) and blocks liquidation for positions on that feed. An attacker with an underwater position can prevent their own liquidation.

Recommendations:

Include `(self.feeds.len() as u32).to_le_bytes()` in the hash before the feeds bytes, pinning the boundary so the two vectors are unambiguously recoverable. The off-chain signer must be updated in lockstep.

Vulnerabilities Medium

Underweight Oracle Consensus

Vulnerability severity:	Medium	Vulnerability probability:	Low	Status:	Resolved

Resolved at commit `103093b8` by adding a defensive `min_agree == 3` path in both price selection functions

`min_agree = 3` Silently Behaves as `min_agree = 2`

In `oracle.rs`, `get_selected_price_by_feed_id_or_name` and `get_selected_price_by_feed_id_or_name_from_inputs_readonly` have an explicit branch only for `min_agree == 1`. For all other values, the same two-block fallback runs.

```
if let (Some(p), Some(s), Some(t)) = (primary_price, secondary_price, tertiary_price) {
    if within_threshold(p.price, s.price, pool_cfg.price_diff_threshold_bps)?
        && within_threshold(p.price, t.price, pool_cfg.price_diff_threshold_bps)?
    {
        return Ok(p);
    }
}

if let (Some(s), Some(t)) = (secondary_price, tertiary_price) {
    if within_threshold(s.price, t.price, pool_cfg.price_diff_threshold_bps)? {
        return Ok(latest_of_two(Some(s), Some(t)).unwrap());
    }
}
```

Since there is no explicit check for `min_agree = 3`, if this were passed in, the 2 of 3 fallback would fire and bypass primary completely. Thus if an attacker could compromise the ChaosLabs and Autonom feeds (smaller and less battle tested), it could allow them to drain pools at will.

This is currently theoretical, as no existing pools have this configuration.

Recommendations:

Add an explicit `min_agree = 3` branch

```
if pool_cfg.min_agree >= 3 {
    if let (Some(p), Some(s), Some(t)) = (primary_price, secondary_price, tertiary_price) {
        if within_threshold(p.price, s.price, pool_cfg.price_diff_threshold_bps)?
            && within_threshold(p.price, t.price, pool_cfg.price_diff_threshold_bps)?
        {
            return Ok(p);
        }
    }
    return Err(AdrenaError::MissingOraclePrice.into());
}
```

Vulnerabilities Medium

Incorrect exponent handling

Vulnerability severity:	Medium	Vulnerability probability:	Info	Status:	Resolved
Resolved commit `61f867fa` by replacing `scale_factor += exponent1;` with `target_power -= exponent1;`					

`math.rs:124` contains a sign error in `checked_decimal_ceil_div`. The `exponent1 > 0` branch uses `target_power += exponent1` where the correct implementation (`checked_decimal_div`, line 72) uses `target_power -= exponent1`. This double-counts the exponent, inflating the result by `10^(2·exponent1)`:

The function is `pub` but currently has zero callers and the bug is not actively exploitable.

Recommendations:

Delete the function and rely on (`checked_decimal_div`) which correctly handles exponents. If a ceiling variant is required in future, fix the sign error (`+= exponent1` - `-= exponent1`)

Missing Slippage Enforcement on Open Position

Vulnerability severity:	Medium	Vulnerability probability:	Medium	Status:	Acknowledged
Acknowledged by Adrena team with: Slippage guards intentionally removed from all 6 position lifecycle functions (open/increase/close for both long and short) to maintain consistency with release/38's design decision (v1.4.1 removed slippage from all instructions). Stop-loss/take-profit execution slippage (`stop_loss_slippage_ok`) preserved — this is SL/TP trigger validation, not user-facing slippage.					

The `price` parameter in `open_position_long/short` and `increase_position_long/short` is documented as "used for slippage protection" but is never compared to the execution price. The position's entry price is set directly from the oracle with no slippage check:

```
// open_position_long.rs:357 - entry price ignores params.price
position.price = token_trade_price.price;
```

The close instructions (`close_position_long/short`) **do** enforce slippage via `MaxPriceSlippage`. Users are protected on exit but not on entry.

The threat is market volatility during Solana network congestion. A user submitting `open_position_long` with a maximum entry could have their transaction delayed, during which a legitimate oracle update moves the price higher. The position opens at the market price, ignoring the user's limit. If the price reverts to the user loses collateral.

Recommendations:

Add the same `MaxPriceSlippage` check that exists in close to the open and increase instructions. For longs: `require!(token_trade_price.price <= params.price, AdrenaError::MaxPriceSlippage)`. For shorts the comparison is inverted.

Vulnerabilities Low

Schedule-Dependent Interest and Funding Inflation via Ceiling Division Rounding

Vulnerability severity:	Low	Vulnerability probability:	Medium	Status:	Acknowledged

Acknowledged by Adrena team with: Per-call inflation is sub-dust and cannot be amplified to material loss at current block times.

Splitting the same elapsed time into multiple custody update calls inflates both cumulative borrow interest and cumulative virtual funding indices because `checked_ceil_div` rounds up each sub-interval independently. Any permissionless operation that touches a custody triggers this rounding on both accumulators simultaneously, allowing an attacker to grief borrowers and funding-paying positions at negligible cost.

The same `checked_ceil_div(dt * rate, 3_600)` pattern appears in two accumulators:

`custody.rs::get_cumulative_interest()`:

```
let cumulative_interest = math::checked_ceil_div(
    (current_time - self.borrow_rate_state.last_update) as u128
        * self.borrow_rate_state.current_rate as u128,
    3_600,
)?;
```

`custody.rs::get_cumulative_funding_indices()`:

```
let delta = math::checked_ceil_div(dt * abs_rate as u128, 3_600)?;
```

Each time `update_borrow_rate` or `update_virtual_funding_rate` is called and persists the result, the ceiling division rounds up independently. A sub-3600-second interval with a nonzero rate contributes a minimum of 1 unit of cumulative index regardless of how short the interval is. With rate = 100,000 and a 1-second interval: `ceil(100_000 / 3_600) = 28`, versus the mathematically correct `100_000 / 3_600 = 27.78`.

Recommendations:

Replace `checked_ceil_div` with floor division in both cumulative interest and cumulative funding calculations. The negligible rounding loss will be dominated by fees and be economically irrational as an attack vector.

Vulnerabilities Low

Hardcoded RPC Authentication Tokens in Source

Vulnerability severity:	Low	Vulnerability probability:	Medium	Status:	Remediated
Remediated at commit `61f867fa` by removing auth tokens from source. They will remain available to anyone with git history access and should be rotated					
!	Fix: Rotate keys				

Three CLI files contain hardcoded RPCPool authentication tokens embedded in URLs

- `test_close_short.ts:7` contains a devnet token.
- `test_close_position.ts:7` contains the same devnet token
- `find_instant_sellers.ts:27` contains a mainnet token

Anyone with repository access can use these endpoints to exhaust rate limits or incur billing on the team's RPC provider account. The mainnet token in `find_instant_sellers.ts` defaults to the hardcoded URL when `process.env.RPC_URL` is unset, meaning the token is used silently in normal operation.

These tokens do not grant access to signing keys or on-chain authority. The risk is limited to RPC provider abuse and denial of service if rate limits are exhausted during operational use.

Recommendations:

Rotate all three tokens immediately. Replace hardcoded URLs with `process.env.RPC_URL` with no default fallback. Add a `.env.example` file documenting required variables.

Vulnerabilities Low

DOS: Residual Unchecked Oracle Math

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Resolved
Resolved at commit `61f867fa` by adding `saturating_sub`					

`oracle.rs` retains plain arithmetic in `OraclePrice::low()` and `OraclePrice::high()`

```
pub fn low(&self) -> Self {
    Self { price: self.price - self.confidence, ... }
}

pub fn high(&self) -> Self {
    Self { price: self.price + self.confidence, ... }
}
```

`overflow-checks = true` is set in `Cargo.toml`, meaning a silent wrap leading to bad debt can no longer occur.

If `confidence > price or price + confidence > u64::MAX`, through corrupted oracle state, a future change to `get_confidence_from_price`, or oracle poisoning/injection via provider data near the `u64` boundary, every instruction calling `low()` or `high()` will abort.

This freezes all close, liquidate and add/remove operations for the affected custody until oracle admin correction.

Recommendations:

Replace the plain operators with saturating arithmetic to prevent the DoS entirely:

```
pub fn low(&self) -> Self {
    Self {
        price: self.price.saturating_sub(self.confidence),
        ..Default::default()
    }
}

pub fn high(&self) -> Self {
    Self {
        price: self.price.saturating_add(self.confidence),
        ..Default::default()
    }
}
```

Vulnerabilities Low

Griefing: Pool-Wide Write-Lock

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Resolved

The project added an early return when there are no fees to distribute, which avoids taking unnecessary write locks in the zero-work case. This removes the specific no-op griefing pattern from the original finding. However, the instruction is still permissionless, so any funded caller can still trigger the fee-distribution path when fees are present and acquire the related write locks.

Permissionless `distribute_fees` can be called as a no-op to write-lock the entire pool and all staking accounts for a full block, blocking every concurrent position and staking operation across all custodies simultaneously.

`pub caller: Signer<'info>` carries no constraint. Any funded wallet can invoke.

The instruction writes to `pool`, `lm_staking`, `lp_staking`, all staking reward token vaults, `staking_reward_token_custody`, and `oracle`. (Due to Solana write locking at deserialization), the locks are held for the entire block even when the instruction returns early:

```
if total_fee_debt_usd == 0 {
    msg!("No fees to distribute");
    return Ok(());  // ← locks held, no state change
}
```

This means an attacker can call `distribute_fees` as a no-op every block to indefinitely write-lock the pool PDA and all staking PDAs simultaneously.

A profitable attack exists in principle by the same mechanism as the liquidation capture finding: the attacker provides pool liquidity (obtaining LP tokens), calls `distribute_fees` each block to hold the write-lock during a volatile window, and submits `liquidate_long` when a target position becomes liquidatable during a blocked block. Because the pool PDA is blocked pool-wide, the attack can target multiple near-liquidation positions in the same pool simultaneously rather than one at a time. Profit is indirect via LP token appreciation from liquidation fees accruing to pool AUM.

Recommendations:

Restrict `caller` to a protocol-controlled address via the admin or a whitelisted keeper keypair registered on `Cortex`.

```
#[account(
    mut,
    constraint = caller.key() == cortex.load()?.admin
        || caller.key() == cortex.load()?.fee_distribution_crank
        @ ErrorCode::InvalidSigner
)]
pub caller: Signer<'info>,
```

Vulnerabilities Low

Griefing: Liquidation Capture

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Acknowledged

Acknowledged by Adrena team with comment: Borrow fee resolution is a protocol-beneficial public good (analogous to liquidation). Restricting to the position owner would leave fees unresolved until the owner acts, degrading pool accounting. The zero-interest early return prevents no-op write-locks, and the attacker gains nothing while spending SOL on transaction fees.

Permissionless `resolve_position_borrow_fees` can be frontrun and defer collateral increases/position closes

`pub signer: Signer<'info>` allows any funded wallet to call functions on this instruction against any open position.

An attacker could identify positions near liquidation threshold and frontrun them with a `resolve_position_borrow_fees` call, which writes to the pool.

Since Solana enforces write lock contention, this will effectively delay the pool, and in a volatile market could in principle force liquidation, preventing holders from removing their position or adding collateral, unless they pay to frontrun the frontrunner.

In the worst case scenario, a griefer could apply this indefinitely (as long as they are willing to bear the gas cost) by paying for this block after block.

A profitable attack does exist in principle: the attacker first provides liquidity to the pool (obtaining LP tokens), then calls `resolve_position_borrow_fees` each block to hold the write-lock during a volatile window. When the price drops below the liquidation threshold during a blocked block, the attacker submits `liquidate_long`; the liquidation fee increases pool AUM, directly appreciating the attacker's LP tokens. The attacker's profit is indirect and requires favorable market conditions — an adverse price move must occur within the blocked window.

Recommendations:

Enforce position owner or a whitelisted admin as the only accounts able to resolve borrow fees

```
#[account(
    constraint = signer.key() == position.load()?.owner
        || signer.key() == cortex.load()?.admin
        @ ErrorCode::InvalidSigner
)]
pub signer: Signer<'info>,
```

Vulnerabilities Low

Update-Time Lock Traps User After `add_collateral`

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Acknowledged

Acknowledged by Adrena team with comment: The lock protects against oracle-stale close exploits and the window is short.

Calling `add_collateral` resets `update_time`, locking the user out of `close_position` for `min_position_update_time_before_close_seconds` (default 120s). During this window the price can continue moving against the user.

`add_collateral` does not affect PnL calculations, fee calculation, open interest, or position sizing, this penalizes the user with no actionable gain.

The timer reset appears to be a blanket pattern applied to all position mutations, and should be retained for financially gameable operations like `increase_position` or `remove_collateral`.

Recommendations:

Remove `update_time` update on `add_collateral`

Vulnerabilities Low

Fee Distribution: Off-Peg Stablecoin Creates Phantom Pool Value

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Resolved

Fee distribution no longer assumes that 1 USD always equals 1 token. The code now converts USD-denominated fee debt into token amounts using the oracle price at distribution time. This removes the phantom-value issue that could appear when the stablecoin traded below peg.

`distribute_fees.rs` sets token transfer amounts directly from USD-denominated fee debt buckets with no price conversion. When USDC ≠ $1.00, zeroing the USD debt while transferring a mismatched token value creates an AUM discrepancy.

Bottom line: Only the above-peg fee recipient timing attack (Scenario A) is economically viable, and its profit is bounded by fee debt size (not pool TVL) and the magnitude of USDC's deviation from peg. The primary harm is to AUM accuracy and LP token pricing fairness, not direct value extraction.

The price gate in `distribute_fees.rs` only enforces a lower bound of `$0.99` with no upper bound. USDC's confidence is hardcoded to zero in `oracle.rs`, so `price.low() = price`.

Three attack scenarios were evaluated:

Scenario	Attack	Viable?	PoC Result
A: Fee recipient timing (above peg)	Call `distribute_fees` when USDC > $1.00; recipients receive ~0.3% excess USD value	Yes	PoC #10: confirmed on $14.41 fee debt
B: LP timing during minor depeg	Call `distribute_fees` during depeg to create phantom AUM, then redeem LP	No	PoC #11: unrealized LP loss exceeds phantom AUM gain
C: Extended depeg accumulation	Accumulate fee debt during prolonged depeg, enter LP, trigger distribution on recovery	No	PoC #13: requires millions at risk during stablecoin crisis

Recommendations:

Convert USD debt to token amounts using the oracle price at distribution time, rather than assuming 1:1:

```
let price = staking_reward_token_price.price;
let one_usd = 10u64.pow(Cortex::PRICE_DECIMALS.into());

let lm_stakers_fee_token_amount =
(pool.lm_fee_debt_usd as u128 * one_usd as u128 / price as u128) as u64;
let protocol_fee_token_amount =
(pool.protocol_fee_debt_usd as u128 * one_usd as u128 / price as u128) as u64;
let referrers_fee_token_amount =
(pool.referrers_fee_debt_usd as u128 * one_usd as u128 / price as u128) as u64;
let manager_fee_token_amount =
(pool.manager_fee_debt_usd as u128 * one_usd as u128 / price as u128) as u64;
```

Additionally, add an upper-bound gate (e.g. `price < $1.01`) to complement the existing lower bound, narrowing the acceptable deviation window symmetrically.

Vulnerabilities Low

Feeless Swap Skips Borrow Rate Update

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Resolved
Resolved at commit `61f867fa` by moving `update_borrow_rate` before the feeless_swap early return					

Internal (feeless) swaps modify `custody.assets.owned` but return early before calling `update_borrow_rate` on either custody. The borrow rate remains based on pre-swap utilization until the next instruction that touches the affected custody.

```
if feeless_swap {
    return Ok(());
}
```

The permissionless `open_or_increase_position_with_swap_long/short` instructions trigger feeless internal swaps via CPI (caller = `transfer_authority` PDA). The subsequent `open_position` CPI updates the **collateral custody** borrow rate, but not the **receiving custody** (the one that gained tokens from the swap).

During the stale window, all interest calculations on the affected custody use `get_cumulative_interest()` which multiplies elapsed time by the stale rate

```
    let cumulative_interest = math::checked_ceil_div(
        (current_time - self.borrow_rate_state.last_update) as u128
            * self.borrow_rate_state.current_rate as u128,
        3_600,
    )?;
```

When a swap adds tokens to a custody, utilization drops, but the old higher rate persists. Borrowers are overcharged for the duration of the stale window. The excess flows to the pool as additional interest, benefiting LPs.

Compound Interaction with L1 (Schedule Dependence): The stale rate and the L1 ceiling rounding finding operate on the same accumulator but are **additive, not multiplicative**. The `checked_ceil_div` rounding error is exactly 0 or 1 per interval regardless of rate magnitude. Repeated swap→lock-in cycles cost 16 BPS per position open ($1.60 minimum) to inflict $0.064 of excess interest per cycle, a 250:1 cost/benefit ratio against the attacker.

Recommendations:

- Call `update_borrow_rate` on both custodies before the feeless early return in `swap.rs`, or add borrow rate updates to `open_or_increase_position_with_swap_long/short` after the swap CPI

Vulnerabilities Low

LP Sandwich: No Structural Hold Period

Vulnerability severity:	Low	Vulnerability probability:	Info	Status:	Resolved

Resolved at commit `61f867fa` by tracking last deposit time and enforcing `require!(current_time > pool.last_lp_deposit_time, AdrenaError::InvalidArgument);`

! Caveat: Since `last_lp_deposit_time` is poolwide, a deposit by any user updates the timestamp, and temporarily blocks all withdrawals. Since slot windows on Solana are ~400ms, the impact in practice should be minimal

LP tokens can be minted and redeemed in consecutive slots with no time-lock enforced.

An attacker can execute a GLP/GMX-style sandwich attack: deposit LP immediately before a large liquidation event and withdraw immediately after, capturing a pro-rata share of any AUM increase.

Currently unprofitable with own capital due to ~28 bps round-trip fees from `add_liquidity` + `remove_liquidity`, and mark-to-market AUM pricing in `get_assets_under_management_usd`. The AUM soft cap further limits maximum sandwich deposit size.

Both defenses are economic. If add/remove liquidity fees is reduced, or if a single liquidation event transfers a sufficiently large fraction of AUM to the pool, the attack becomes profitable.

Flash loan composition note: Since `add_liquidity` and `remove_liquidity` have zero cooldown, they can be composed with external Solana flash loan protocols within a single transaction via Jito bundles.

Recommendations:

Add a minimum LP hold period (e.g. matching `min_position_open_time_seconds`) to provide structural sandwich prevention independent of fee economics. Even a single-slot delay would defeat flash loan composition entirely, since the loan must be repaid within the same transaction. This would also partially mitigate H1 (VFR AUM mispricing) by preventing instant LP redemption against inflated AUM during funding imbalance windows.

Griefing: Deprecated Permissionless IDL

Vulnerability severity:	Low	Vulnerability probability:	Info	Status:	Resolved

Resolved at commit `61f867fa` by returning `InvalidInstructionData.into()`

`genesis_stake_patch.rs` appears as callable in IDL even though it is deprecated

The instruction is fully commented out and silently returns `Ok(())`. The instruction is permissionless and lists as mutable staking, user_staking, cortex, staking_reward_token_vault, staking_lm_reward_token_vault, lm_token_mint PDAs.

This means a malicious actor could grief those PDAs by triggering a lock on this publicly callable no-op. Since Solana enforces write lock contention, in principle an attacker could grief these PDAs indefinitely, so long as they were willing to pay the transaction fees. No profit path has been identified. In Fidesium's estimation this represents purely a griefing vector.

Additionally, third party integrators who identify it in the IDL could call it, receive an `Ok(())`, and suffer downstream issues causing communications spam and loss of community

Recommendations:

Remove the instruction from the IDL

Vulnerabilities Low

Wrapping Overflow in Financial Calculation

Vulnerability severity:	Low	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by adrena team with comment: `collected_fees` and `volume_stats` are observability-only accumulators with no control flow dependency. Wrapping after ~18.4 exaUSD is operationally unreachable.

`pool.rs` uses wrapping computation for numbers with financial impact.

`accrue_fee_debt` uses `wrapping_add` for debt accumulator fields.

```
self.manager_fee_debt_usd = self.manager_fee_debt_usd.wrapping_add(manager_fee_usd);
...
self.lm_fee_debt_usd = self.lm_fee_debt_usd.wrapping_add(lm_fee_usd);
...
self.protocol_fee_debt_usd = self.protocol_fee_debt_usd.wrapping_add(protocol_fee_usd);
...
self.referrers_fee_debt_usd = self.referrers_fee_debt_usd.wrapping_add(referrer_fee_usd);
```

And then in `distribute_fees.rs`

```
let total_fee_debt_usd = pool.lm_fee_debt_usd
        + pool.protocol_fee_debt_usd
        + pool.referrers_fee_debt_usd
        + pool.manager_fee_debt_usd;
```

Plain `+` in `distribute_fees.rs:197-200` panics on overflow, and will permanently revert DoSing the protocol, requiring a migration to fix

Since an individual field will generally wrap before the four field sums exceeds `u64::MAX` AUM inflation will happen first and DoS is improbable

The `cumulative_*` fields are read only logging fields and use `wrapping_add` intentionally.

This risk is theoretical since the threshold is undistributed fees of ~18.4T$

Recommendations:

Replace with `saturating_add`/`checked_add` in `accrue_fee_debt`

Replace with `saturating_add`/`checked_add` in `distribute_fees.rs`

Convert `total_fee_debt_usd` in `distribute_fees` from `u64` to `u128`

`checked_add` is a safer choice but will require manually handling errors.

Vulnerabilities Low

Governance Lacks Participation Quorum and Execution Timelock

Vulnerability severity:	Low	Vulnerability probability:	Low	Status:	Acknowledged

Acknowledged by Adrena team with comment: On-chain governance configuration (SPL Governance program state), not Adrena source

The AdrenaDAO governance (`HgeoVqTTMQ9K5GZAUpPKaz5PS8Rn55yR5e5SwmB3DbKB`) is configured with a `YesVotePercentage(45%)` threshold, which requires 45% of *deposited* votes to approve — not 45% of total token supply. There is no minimum participation quorum, no execution timelock (`transactions_hold_up_time = 0`), and no council or veto body.

On-chain analysis of 272 historical proposals reveals that governance is effectively controlled by a single ~800M token voting bloc (~47% of the 1.72B supply). When this bloc votes, proposals pass unanimously. When it does not participate, proposals are defeated — not by opposition, but by insufficient votes. Only 4 proposals in the entire history received meaningful "No" votes.

Under current conditions this is not exploitable: the dominant bloc consistently participates at 46–49% of supply on legitimate proposals. The absence of a participation floor means that if this bloc were ever inactive, a much smaller holder could pass proposals with arbitrarily low participation.

The zero-second execution timelock compounds this: an approved malicious proposal executes immediately with no window for the community to react, withdraw funds, or counter-vote.

Empirical impact analysis of hypothetical quorum thresholds against all 210 historically passing proposals:

```
Quorum     Legitimate proposals blocked
──────     ────────────────────────────
5–30%      0 / 210  (0%)
35%        43 / 210 (20%)
40%        52 / 210 (25%)

Historical passing proposal participation:
Min:     31.4%
P25:     42.4%
Median:  47.2%
Max:     73.7%
```

Recommendations:

- Introduce a **20–25% participation quorum**. This would have blocked zero legitimate proposals historically while preventing low-participation attacks. A 25% threshold provides a 6 percentage point buffer below the historical minimum (31.4%).
- Set `transactions_hold_up_time` to **24–48 hours**. This gives the community a reaction window after proposal approval before execution, at negligible governance cost.
- Consider enabling a **council or veto role** as a guardian mechanism, particularly for high-impact actions such as program upgrades and admin transfers.

Vulnerabilities Info

Deprecated Instruction: Permissionless Account Corruption

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved
Resolved at commit `61f867fa` by valdating PDA					

`migrate_pool_v37_to_v38.rs` does not validate PDA seed or discriminator.

The successor instruction `migrate_pool_v38_to_v39` correctly enforces `has_one = admin`, PDA seeds, bump, and Anchor's `AccountLoader` discriminator check

Once both migration have run, an attacker would need to identify another Adrena program-owned account with data length exactly equal to `PoolLegacy::LEN`.

Recommendations:

Once migration has been run, remove `migrate_pool_v37_to_v38` from `lib.rs` entirely.

Referrer Fee Stripping: Optional Account Omission

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged
Acknowledged by Adrena team with comment: Omitting the referrer account only forfeits the referrer's share (redirected to LP/protocol). It does not create a pool accounting discrepancy.					

Referrer fee accrual in `pool.rs:accrue_fee_debt` depends on the caller-supplied `has_referrer` boolean, which is derived from whether optional `user_profile` and `referrer_profile` accounts are passed.

When a caller omits both optional profile accounts

- The referrer validation block is skipped entirely — `has_referrer` defaults to `false`
- `accrue_fee_debt` routes zero fees to `referrers_fee_debt_usd`
- The referrer's share is absorbed by the `protocol_fee_debt_usd` bucket
- The referrer's `UserProfile.claimable_referral_fee_usd` is never credited

`referrer_fee_share_bps = 0` is set on all deployed pools (set during `add_pool_part_one` and confirmed in MIGRATION.md). **The referrer fee system is currently turned off entirely**, making this finding zero-impact in production today.

The total fee charged to the trader is unchanged. Only the internal split between protocol and referrer buckets is affected. No user funds are at risk.

On open/increase paths, omitting `user_profile` triggers `MAX_FEE_MULTIPLIER = 4`, making self-exploitation on entry economically irrational.

If the protocol enables non-zero `referrer_fee_share_bps` in the future, any permissionless keeper running liquidations or borrow-fee resolution would systematically strip referrer fees from every transaction, and traders could selectively suppress referrer payouts on close.

Recommendations:

Derive the referrer relationship from on-chain state (look up the `user_profile` PDA from the position owner and read `referrer_profile`) rather than trusting the caller to pass optional accounts. Alternatively, make `user_profile` mandatory when one exists on-chain for the position owner.

Vulnerabilities Info

Deprecated Callable IDL Instruction

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved
Resolved at commit `61f867fa` by returning `InvalidInstructionData.into()`					

`patch_custody_locked_amount.rs` appears as callable in IDL even though it is deprecated.

The instruction is fully commented out and silently returns `Ok(())`

There is no known active exploit path but it does unnecessarily widen the attack surface.

A social engineering attack is possible (since it appears in the IDL, a malicious frontend could present it as a required call to get an admin signature).

Recommendations:

Remove the instruction from the IDL

Shared PDA Namespace Between accounts

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved
Resolved at commit `61f867fa` by adding runtime collision detection					

Both `add_custody` and `add_synthetic_custody` derive PDAs from `["custody", pool, seed]` with an arbitrary `[u8; 32]` seed.

A real custody whose mint pubkey equals a synthetic custody's seed would resolve to the same address. Anchor's `init` constraint prevents double-initialization.

Anchor's `init` constraint prevents double-initialization, so whichever is created first occupies the slot and the second silently fails

The shared namespace is a natural consequence of both types being stored as the same `Custody` struct, distinguished at runtime via the `is_synthetic` field.

A malicious or erroneous admin could intentionally or accidentally block a token mint. A malicious or compromised admin would have far more destructive options, and an admin error could be recovered from by using a different pool or a different seed.

Serving the incorrect custody type to an instruction would require an admin to further bypass the `is_synthetic check`

Recommendations:

Consider using distinct seed prefixes (e.g. `["custody", pool, seed]` vs `["synthetic_custody", pool, seed]`) to enforce type separation at the PDA layer.

Vulnerabilities Info

Unchecked Subtraction

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by adrena with comment: A panic is the correct response to impossible state. `overflow-checks = true` in `Cargo.toml` ensures runtime detection.

`Position::subtract` uses `-=`, which will revert with a generic panic due to `overflow-checks = true`

This is not currently exploitable since `position` is never modified between clone and subtract, structurally maintaining the `position.field >= position_to_close.field` constraint.

A future change mutating position, to, for instance, resolve borrow fees in line, or deducting a new fee type, could turn this into a DoS vector.

Recommendations:

Consider using `checked_sub` to make the error explicit during debugging and development.

```
self.size_usd = self.size_usd
    .checked_sub(position.size_usd)
    .ok_or(AdrenaError::MathOverflow)?;
```

Permissionless Migration: Nickname Front-Run and Snipe

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by adrena with comment If called permissionlessly, nickname must be either the same as it was, or match a specific pattern

`migrate_user_profile_from_v1_to_v2` accepts any `caller: Signer` with no owner gate, allowing a third party to consume the one-shot V1-V2 migration for any legacy profile.

A non-owner caller is constrained to setting the nickname to either the existing V1 nickname or a generic `MonsterXXX` autogenerated pattern. By choosing `MonsterXXX`, the attacker forces the owner off their original nickname without creating a reservation PDA for it. An accomplice can then claim the original nickname on a fresh profile via `init_user_profile`, achieving on-chain identity impersonation. The victim cannot re-migrate (one-shot already consumed) and must pay the `CHANGE_NICKNAME_TAX` (500 ADX burn via `edit_user_profile_nickname`) to obtain any non-`MonsterXXX` name, but cannot reclaim the sniped nickname since its PDA is now occupied. 343 V1 profiles still exist on-chain at the time of this assessment, so the migration window has not closed and the attack surface remains live.

No profit vector has been identified. Nicknames are purely cosmetic on-chain. Referral links, fee distribution, trading, and liquidation logic all use pubkeys, not display names. The impact is limited to social impersonation on frontend leaderboards and a forced nickname-change tax payment.

Recommendations:

Consider adding an `owner == caller` constraint to the migration instruction, or restricting the non-owner path to only preserve the existing V1 nickname (removing the `MonsterXXX` alternative). Alternatively, batch-migrate the remaining 343 V1 profiles server-side using their original nicknames to close the window entirely.

Vulnerabilities Info

Fresh Pool LP Price Can Transiently Round to Zero

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Self-correcting and only affects the informational AUM view, not actual token balances or LP redemption math.

When a position is opened on a freshly bootstrapped pool, `open_position` accrues fee debt (manager, protocol, LM, referrer shares) without calling `distribute_fees`. Since `get_assets_under_management_usd` subtracts all fee debt buckets from AUM, this causes AUM to dip below LP token supply. On a pool with ~$25M AUM (represented as ~24.9 trillion in 6-decimal USD), the integer division `aum / lp_supply` truncates to **0**.

Fresh pools bootstrap with an AUM-to-supply ratio of approximately 1:1, leaving zero headroom. Any fee-generating operation pushes AUM below supply because:

- `open_position` accrues fee debt but does **not** distribute fees
- `distribute_fees` is a separate instruction, called only within `add_liquidity`, `remove_liquidity`, and position close/liquidation transactions
- Between accrual and distribution, fee debt persists on-chain and deflates reported AUM

On mature pools this is a non-issue: accumulated LP fees (70% share) push AUM well above supply, providing ample buffer.

Any external system (UI, dashboard, bot) that computes `aum / supply` as an integer could display LP tokens as worthless during the fee-debt window on a fresh pool. This is cosmetic and self-corrects once `distribute_fees` runs or trading fees accumulate.

Recommendations

1. **Do nothing (acceptable):** The issue is cosmetic, non-exploitable, and self-corrects as the pool matures. Document that LP price can transiently dip below 1.0 on fresh pools.
2. **Distribute fees inline:** Call `distribute_fees` from within `open_position` to prevent fee debt from persisting across transactions. Adds ~200K compute units per trade.
3. **Seed initial LP price above 1.0:** Mint fewer LP tokens than the USD value on the first deposit (e.g., start at LP price = 1.10) to create a permanent buffer against fee-induced AUM dips.

Authority Convergence: BPF Upgrade Authority and Cortex Admin

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Off-chain configuration issue. Not verifiable or addressable from program source code.

Both the BPF Loader upgrade authority and the on-chain `Cortex.admin` are set to the same address: the DAO native treasury `7VzEXYvGmLg3tdVuFuGFQdr7GP5tutTUt8EcTGHvG8Ev`.

Recommendations:

Consider separating the two authorities so that a program upgrade requires a different approval path than admin configuration changes. For example:

- Assign the BPF upgrade authority to a higher-threshold multisig (e.g., Squads) or a separate governance with a stricter quorum, while leaving `Cortex.admin` under the existing DAO treasury for day-to-day operations.
- Alternatively, introduce a two-step `propose_admin` / `accept_admin` pattern with a timelock on `set_admin`.

Vulnerabilities Info

Low Proposal Creation Threshold Enables Governance Spam

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: On-chain governance state managed by SPL Governance program. Not configurable from Adrena program source code.

The governance configuration sets `min_community_weight_to_create_proposal` to 1,000,000 tokens, which represents 0.058% of the total 1.72B token supply. This is a very low barrier for proposal creation.

An attacker or griefer with a modest token holding could flood the DAO with proposals in order to:

- Induce **voter fatigue**, reducing participation on legitimate proposals over time. This directly increases the feasibility of a low-turnout governance attack.
- **Bury a malicious proposal** among dozens of benign or nonsensical ones, making it less likely to receive scrutiny before the 3-day voting window closes.
- **Degrade the governance UX** on the Realms interface, making it harder for legitimate participants to find and engage with real proposals.

On-chain evidence shows this is not purely theoretical: several historical proposals appear to be tests or dust (e.g., "TEST TEST 2" with 0.0% participation, proposals with titles like "add sol custody" receiving only 9,810 votes).

Recommendations:

Raise `min_community_weight_to_create_proposal` to a level that imposes meaningful economic cost without disenfranchising legitimate participants. A threshold of **5,000,000–10,000,000 tokens** (0.3–0.6% of supply) would increase the cost of spam by 5–10x while remaining well within reach of any stakeholder with a genuine governance interest. The original mainnet deployment used 5,000,000 per `cli/README-MAINNET.md:409`.

Dead Code: Partially Unused SPL Governance CPI Adapter

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Rewritten as vendored types in release/39 (spl-governance crate incompatible with Solana v2/Anchor 0.31). Unused instruction builders retained for potential future governance operations with `#[allow(dead_code)]`

The file `adapters/spl_governance_adapter.rs` defines 5 CPI wrapper functions for the SPL Governance program which are never used

```
set_governance_delegate
withdraw_governing_tokens
cast_vote
relinquish_vote
revoke_governing_tokens
```

Recommendations:

Remove the 5 unused adapter functions.

Vulnerabilities Info

No runtime invariant between dual OI tracking systems

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: The two systems serve different purposes and divergence is bounded by position lifecycle operations that update both paths.

Funding rates and AUM/PnL compute distinct Open Interest

The protocol computes `trade_stats.oi_long_usd / oi_short_usd` and `long_positions.size_usd / short_positions.size_usd` separately.

While these do not and cannot currently diverge (see positive observations), the impact in the event of a future code change could be catastrophic. An attacker could simultaneously bypass OI caps, extract excess funding from existing traders, and hide real pool exposure from AUM

Particularly concerning is the `open_positions += 1` hack in `close_position_long.rs:547` to handle partial closes and the equivalent logic in synthetic position accounting, which an otherwise qualified engineer could remove as "a hack".

Recommendations:

Add an onchain invariant check after every OI mutation. `require!(custody.trade_stats.oi_long_usd == custody.long_positions.size_usd, AdrenaError::InvariantViolation);` in all four mutation paths. Apply equivalently for short OI.

Stale Funding Rate After Config Downgrade

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Admin config changes are rare governance operations. Operators should settle open positions or call `update_virtual_funding_rate` immediately after parameter changes.

`set_custody_virtual_funding` overwrites `custody.virtual_funding` config parameters but does not recalculate or clamp the live `current_rate_long_to_short` stored in `custody.virtual_funding_state`.

When an admin lowers `max_hourly_funding_rate`, the previously-computed rate remains active until the next user transaction triggers `update_virtual_funding_rate()`.

During the window between the config change and the next funding rate update, positions settling funding use the stale over-cap rate. This is admin-only and self-corrects on the next `update_virtual_funding_rate()` call. The practical impact is limited to a transient overshoot in funding charges during an admin operation.

Recommendations:

Call `custody.update_virtual_funding_rate(clock.unix_timestamp)` after writing the new params, or clamp the stored rate: `current_rate_long_to_short = current_rate_long_to_short.clamp(-new_max, new_max)`.

Vulnerabilities Info

Core Dependencies outdated

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved
Resolved at commit `103093b8` by updating to Anchor 0.31.1, Solana 2.1.0, and eliminating the custom fork					

The Solana SDK and Anchor are out of date with latest versions.

Recommendations:

Upgrade Anchor and Solana SDK at your earliest convenience. This will also collapse duplicate versions of other dependencies, eliminate a custom fork requirement, and resolve all flagged advisories. See Appendix K for details.

Short Borrow Interest AUM Asymmetry

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged
Acknowledged by Adrena team with comment: The asymmetry is bounded by position lifecycles and borrow fee resolution. Impact is sub-basis-point on typical pools.					

The AUM calculation treats long and short borrow interest asymmetrically. Long unrealized interest is subtracted from AUM continuously (`pool_value_sub_usd += long_pnl.borrow_fee_usd`, pool.rs), but short unrealized interest is not

```
pool_value_sub_usd += long_pnl.borrow_fee_usd as u128;
```

Short borrow interest only appears in AUM after `resolve_position_borrow_fees` crystallizes it into `prepaid_interest_usd`.

Recommendations:

Add `pool_value_sub_usd += short_pnl.borrow_fee_usd` alongside the long subtraction for accounting correctness, even though the practical impact is limited.

Vulnerabilities Info

Liquidity Quote View Overstates LP Tokens

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Divergence is informational only. Actual mint uses the authoritative on-chain calculation.

The `get_add_liquidity_amount_and_fee` view instruction returns more LP tokens than `add_liquidity` actually mints on the same pool state. The view divides by `pool.aum_usd` (cached), while execution divides by `(pool_amount_usd + lp_fee_usd)` (fresh, larger denominator)

```
lp_amount = token_amount_usd * supply / pool.aum_usd

lp_amount = token_amount_usd * supply / (pool_amount_usd + lp_fee_usd)
```

Recommendations:

Align the view denominator with execution: use `(pool_amount_usd + lp_fee_usd)` instead of `pool.aum_usd`, or document that the view returns an upper-bound estimate.

No Insurance Fund or Bad Debt Tracking

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Acknowledged

Acknowledged by Adrena team with comment: Adding an insurance fund requires new state accounts and economic design beyond current scope.

When a liquidated position's loss exceeds its collateral and accrued fees, the protocol silently absorbs the deficit into the pool. The `get_exit_position_numbers` function in `pool.rs` computes two deficit values: `deficit_fee_usd` (uncollected fees) and `deficit_pool_usd`. These are never persisted on-chain.

The loss is absorbed by decrementing `custody.assets.owned` during position closure, which reduces AUM and dilutes LP token value. There is no dedicated insurance fund, no cumulative bad debt counter, and no circuit breaker to halt trading if socialized losses exceed a threshold.

Bad debt arises from the gap between liquidation eligibility (position exceeds `max_leverage`) and actual liquidation execution. Oracle price gaps, block latency, and continued borrow fee accrual can all push a position further underwater during this window. While `saturating_sub` on the AUM calculation prevents a negative total, the combination of untracked bad debt events and the fee debt `wrapping_add` overflow (L12) can mask the true magnitude of LP losses.

Compound Interaction with L12 (Wrapping Overflow): If fee debt fields wrap to zero via `wrapping_add`, the accumulated bad debt becomes invisible — `distribute_fees` sees near-zero debt and distributes nothing, while the pool's `assets.owned` has already been decremented. The combination means LP losses from bad debt are both untracked (I16) and potentially hidden by the overflow (L12). The v38 migration freeze window amplifies this: blocked liquidations allow positions to accrue deeper bad debt, which is then absorbed silently on resume.

Recommendations:

Add a cumulative `bad_debt_usd: u64` field to `Pool` state, incremented by `deficit_pool_usd` on every liquidation that produces a shortfall. This gives LPs and governance visibility into socialized losses without changing the absorption mechanism. Consider adding an admin-configurable bad debt threshold that pauses new position opens when breached.

Vulnerabilities Info

No Verified Build Pipeline or Source Integrity CI

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved
Resolved at commit `61f867fa` by adding `Dockerfile.verifiable` and locally reproducible build scripts					
!	Minor gap: No Trojace Source CI check added				

No deterministic build configuration (Dockerfile, Solana Verified Builds) exists in the repository, and the binary is produced on a developer's local machine. Without a verified build pipeline, users and auditors cannot independently confirm that the deployed bytecode matches source.

The on-chain program at `13gDzEXCdocbj8iAiqrScGo47NiSuYENGsRqi3SEAwet` (last deployed slot 402672276, 2026-02-25) was compared against a deterministic Docker build of branch `release/37_4` at commit `5242fcdb` using the `solana-verify` toolchain (`solanafoundation/solana-verifiable-build`, Solana v1.18.22, platform-tools v1.41).

Result: bit-exact match. All 3,031,072 bytes of program content are identical. The remaining 1,314,880 bytes in the on-chain account are Solana zero-padding. SHA-256 of program content: `ce9707cbb1156584a953bb831a51aa97c048b34515c2bf9f579b2bc6fa96df6f`. See Positive Observations: *On-Chain Binary Verified Against Source*.

The project does not have a pipeline that makes this reproducible on an ongoing basis. Attack scenarios that exploit this gap for future deployments include: a compromised/malicious developer machine patching the `.so`, dependency substitution at build time, and compiler-level tampering.

Separately, while the source is currently clean of Trojan Source attacks (see Positive Observations: *Source Code Free of Trojan Source and Homoglyph Attacks*), there is no CI enforcement to prevent future introduction of invisible Unicode characters — bidirectional overrides, zero-width joiners, or homoglyphs — that could alter program semantics while passing human code review.

Recommendations:

1. Adopt Solana Verified Builds. Configure `solana-verify` to produce deterministic Docker-based builds, enabling anyone to independently confirm that the deployed bytecode matches the source at a specific commit. The ad-hoc verification performed during this audit confirmed the current deployment is clean, but future deployments need the same guarantee without auditor intervention.

2. Add a Trojan Source CI check. Add a pipeline step that fails on any non-ASCII or bidirectional Unicode in program source:

```
grep -rP '[\x{200B}-\x{200F}\x{202A}-\x{202E}\x{2060}-\x{2064}\x{FEFF}]' programs/adrena/src/ && exit 1 || true
```

This is a zero-cost preventive control that maintains the clean state documented in the positive observations.

Vulnerabilities Info

Fee Debt Accrued at Mid Price, AUM Assets Valued at Low Price

Vulnerability severity:	Info	Vulnerability probability:	Info	Status:	Resolved

Fee accrual in add_liquidity, remove_liquidity, and swap now uses token_price_low.get_asset_amount_usd() instead of token_price.get_asset_amount_usd() (mid-price). This aligns fee debt denomination with how get_assets_under_management_usd values custody.assets.owned (also at .low()), eliminating the per-operation AUM leak of fee_tokens * price * confidence_ratio. Position instructions were not affected as they already used conservative pricing via get_exit_position_numbers.

There is a fee debt price inconsistency between `custody.assets.owned` and debt

`let non_lp_fee_usd = token_price.get_asset_amount_usd(non_lp_fee_amount, custody.decimals)?` is priced at oracle mid

`let token_amount_usd = token_price.low().get_asset_amount_usd(custody.assets.owned, custody.decimals)?` is priced at oracle low/p>

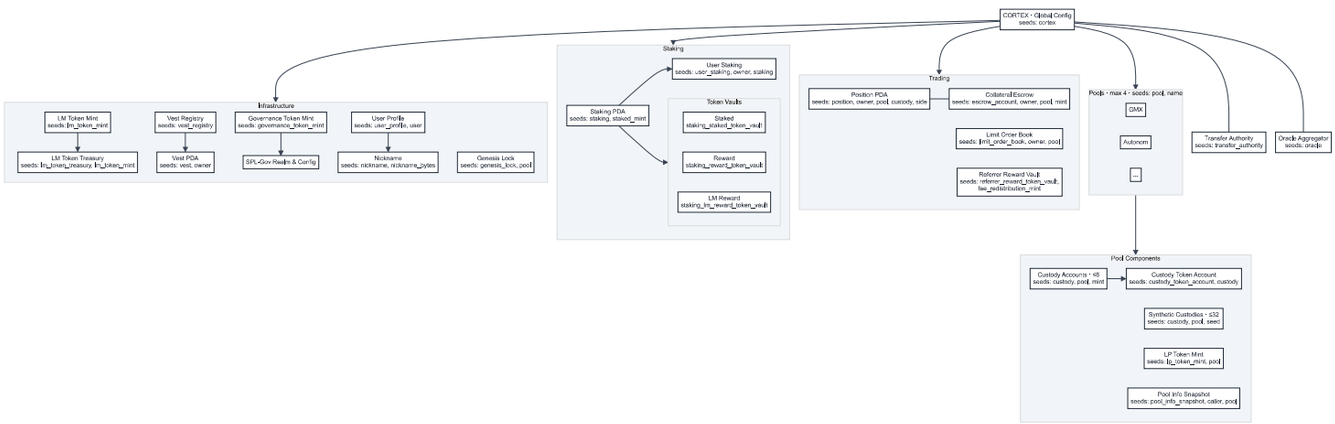
This results in a leak worth non_lp_fee_tokens * price * 0.0025 per operation, assuming confidence stays at 0.25%

This is not exploitable, as an attacker would lose out to fees

Recommendations:

Price liquidity deposits and position open/close/increase/swap at low. This is the conservative direction and favors the pool slightly. `let non_lp_fee_usd = token_price.low().get_asset_amount_usd(non_lp_fee_amount, custody.decimals)?;`

APPENDIX A: Architecture Diagram



APPENDIX B: PDA Derivation Map

Account	Seeds	Notes
Cortex	["cortex"]	Singleton global config
Transfer Authority	["transfer_authority"]	Signs all token transfers
LM Token Mint	["lm_token_mint"]	ADX governance token
Pool	["pool", name]	Up to 4 pools
LP Token Mint	["lp_token_mint", pool]	Per-pool LP token
Custody (real)	["custody", pool, mint]	Real asset custody
Custody (synthetic)	["custody", pool, seed]	Synthetic stock custody
Custody Token Account	["custody_token_account", custody]	Token vault
Position	["position", owner, pool, custody, side]	1 per user/pool/custody/side
Oracle	["oracle"]	Singleton oracle store
Staking	["staking", staked_mint]	LM or LP staking
UserStaking	["user_staking", owner, staking]	Per-user staking
Staked Token Vault	["staking_staked_token_vault", staking]	
Reward Token Vault	["staking_reward_token_vault", staking]	
LM Reward Token Vault	["staking_lm_reward_token_vault", staking]	
LimitOrderBook	["limit_order_book", owner, pool]	Per-user order book
GenesisLock	["genesis_lock", pool]	Campaign state
Vest	["vest", owner]	Token vesting
Referrer Reward Vault	["referrer_reward_token_vault", mint]	Fee redistribution
UserProfile	["user_profile", owner]	Social/achievement
VestRegistry	["vest_registry"]	Singleton that tracks all Vest PDAs
LM Token Treasury	["lm_token_treasury", lm_token_mint]	Holds LM token reserves
Governance Token Mint	["governance_token_mint"]	SPL-Gov voting token
Pool Info Snapshot	["pool_info_snapshot", caller, pool]	per-user pool state cache
Collateral Escrow	["escrow_account", owner, pool, mint]	per-user collateral holding
User Profile Nickname	["nickname", nickname_bytes]	unique nickname reservation

APPENDIX C: Token Flows

```
================================================================
   1. POSITION LIFECYCLE
================================================================

    USER                          CUSTODY                       POOL STATE
    ====                          =======                       ==========

    open_position_long/short
    [collateral] ──────────────► token_account
                                  ├─ assets.owned += collateral
                                  └─ assets.locked += locked_amount

    open_with_swap
    [token A] ─► custody A ── internal swap (feeless) ─► custody B
                                                          └─ position opened here

    add_collateral
    [tokens] ──────────────────► token_account
                                  └─ assets.owned += amount
                                     (no change to size/leverage/fees)

    close_position (profit)
    [payout] ◄────────────────── token_account
                                  ├─ assets.owned -= payout
                                  ├─ assets.locked -= locked_amount
                                  └─ fees accrued ──────────► pool.*_fee_debt_usd

    close_position (loss)
                                  token_account
                                  ├─ assets.owned keeps loss portion
                                  ├─ assets.locked -= locked_amount
                                  └─ fees accrued ──────────► pool.*_fee_debt_usd

    liquidate_long/short (keeper-triggered)
                                  Same as close, but:
                                  └─ liquidation_fee accrued ─► pool.*_fee_debt_usd

    resolve_borrow_fees (during close)
                                  position.paid_interest_usd set
                                  └─ interest stays in custody (already "distributed")
```

APPENDIX C: Token Flows

```
═══════════════════════════════════════════════════════════════════
 2. FEE DISTRIBUTION
═══════════════════════════════════════════════════════════════════

  Fee Sources                    Pool Debt Buckets          distribute_fees()
  ===========                    =================          =================

  open_position fee ──┐
  close_position fee ─┤          pool.lm_fee_debt_usd ──────► LM Staking Reward Vault
  liquidation fee ────┼────────► pool.protocol_fee_debt_usd► Protocol Fee Recipient
  swap fee ───────────┤          pool.manager_fee_debt_usd ► Manager Fee Recipient
  add/remove liq fee ─┘          pool.referrers_fee_debt_usd► Referrer Reward Vault

                                 LP share: never transferred — implicitly retained
                                 in staking_reward_token_custody (USDC)

  Note: paid_interest_usd (from resolve_borrow_fees) is subtracted
        from fee_usd_to_distribute before accrual to avoid double-counting.

  Note: All distributions paid from staking_reward_token_custody
        (stable token). Non-stable fees converted via internal swap.
```

```
═══════════════════════════════════════════════════════════════════
 3. LIQUIDITY, STAKING & VESTING
═══════════════════════════════════════════════════════════════════

  USER                           PROTOCOL                    USER
  ====                           ========                    ====

  [tokens] ─── add_liquidity ──► Custody token_account
                                   LP tokens minted ────────► [LP tokens]
  [LP tokens] ─ remove_liq ────► LP tokens burned
                                   Custody pays out ────────► [tokens]

  [ADX] ──── add_stake ────────► Staking Staked Token Vault
                                   Governance power synced
                claim_stakes ──► Staking Reward Token Vault
                                   USDC rewards ────────────► [USDC]
                remove_stake ──► Staked Token Vault
                                   ADX returned ────────────► [ADX]
                claim_vest ────► LM Token Treasury
                                   ADX minted ──────────────► [ADX]

  [token A] ─── swap ──────────► Custody A - Custody B
                                   (minus swap fees)  ──────► [token B]

  [LP Token Mint/Burn]
      add_liquidity: user deposits ──► custody, receives LP tokens (minted)
      remove_liquidity: user burns LP ──► receives tokens from custody
  [LM Token Minting]
      mint_lm_tokens_from_bucket: admin mints from bucket allocations
      Staking emissions: resolve_staking_round mints LM rewards
```

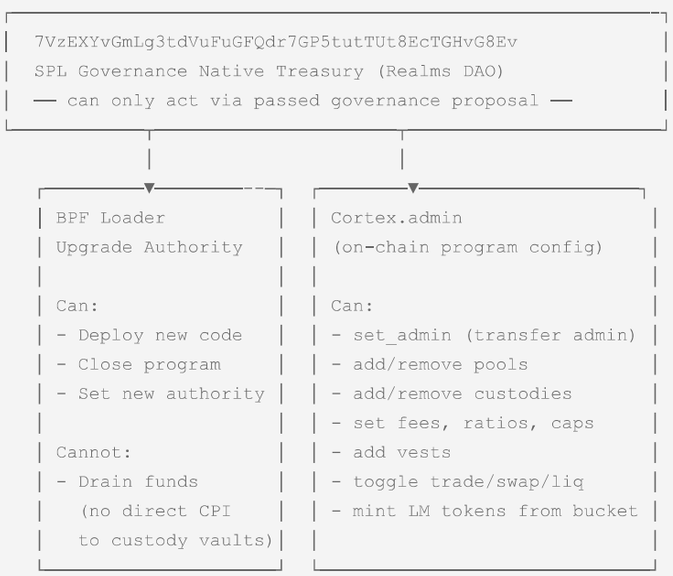
APPENDIX D: Onchain State

Field	Value
Program ID	13gDzEXCdocbj8iAiqrScGo47NiSuYENGsRqi3SEAwet
Loader	BPFLoaderUpgradeab1e11111111111111111111111
Upgrade Authority	7VzEXYvGmLg3tdVuFuGFQdr7GP5tutTUt8EcTGHvG8Ev
Mutability	Upgradeable

Upgrade Authority is a Realms DAO treasury

APPENDIX E: Authority Structure

AUTHORITY ROLES



Role	Storage	Properties	Timelock
Admin	`cortex.admin`	Single signer. Controls all 45 admin instructions: fee config, trading flags, oracle patches, custody/pool management, token minting, vesting. No multisig requirement.	None
Transfer Authority	PDA `["transfer_authority"]`	Program-controlled. Owns all custody token accounts and vault token accounts. Signs token transfers on behalf of the protocol. Cannot be changed by admin.	N/A
Protocol Fee Recipient	`cortex.protocol_fee_recipient`	Receives protocol's share of trading fees after `distribute_fees`. Set by admin via `set_protocol_fee_recipient`.	None
Manager Fee Recipient	`pool.manager_fee_recipient`	Per-pool. Receives manager's share of fees on Autonom pools. Set by admin via `set_pool_fee_config`.	None
Governance	`cortex.governance_program`	Informational only. Tracks voting power via staked tokens. Does NOT gate any admin actions or protocol operations.	N/A
Liquidator	Any signer	Permissionless. Can call `liquidate_long/short` on any position exceeding max leverage. Receives liquidation fee.	N/A
SL/TP Executor	Any signer	Permissionless. Can trigger `close_position` when stop-loss or take-profit price is reached. Receives automation execution fee.	N/A

APPENDIX F: Admin Powers

ADMIN POWERS

Category	Instructions	State Modified	Risk
Authority Transfer	`set_admin`	`cortex.admin`	Irreversible single-step transfer of all admin powers. No confirmation or timelock.
Fee Control	`set_pool_fee_config`, `set_all_pools_fee_shares`, `set_protocol_fee_recipient`	Pool fee split ratios (LP, LM, protocol, manager, referrer), pool type, fee recipient wallets	Can redirect up to 100% of trading fees to any wallet. No cap on individual share beyond 10,000 BPS total.
Trading Freeze	`set_pool_allow_trade`, `set_pool_allow_swap`, `set_custody_allow_trade`, `set_custody_allow_swap`	Pool and custody trading/swap flags	Can freeze all trading and liquidity operations instantly. Existing positions cannot be opened or closed (but can still be liquidated).
Exit Fee / Time Locks	`set_pool_position_exit_fee_config`	`position_exit_fee_config`: min open time, min update time, tier multipliers	Can set extreme `min_position_open_time_seconds` to lock users out of closing positions. Liquidation is unaffected (see l6_exit_fee_mev_interaction tests).
Oracle Control	`set_pool_oracle_config`, `patch_custodies_oracles`, `init_oracle`	Oracle provider selection, multi-oracle config (min_agree, staleness), custody oracle feeds	Can change price sources, reduce `min_agree` to 1, extend staleness window. Could enable stale or manipulated price acceptance.
Custody Management	`add_custody`, `add_synthetic_custody`, `remove_custody`, `set_custody_config`	Custody list, fees, pricing params, borrow rates, oracle assignment	Can add/remove tradeable assets, change per-custody fees and leverage limits. `remove_custody` affects ratio calculations for remaining custodies.
Position Size Limits	`set_custody_max_cumulative_long_position_size_usd`, `set_custody_max_cumulative_short_position_size_usd`	Per-custody OI caps	Can restrict or expand maximum open interest per asset. Setting to 0 disables the cap.
Pool Lifecycle	`add_pool_part_one/two`, `remove_pool`, `set_pool_liquidity_state`, `set_pool_aum_soft_cap_usd`	Pool creation/removal, liquidity state machine, AUM cap	Can create new pools, decommission existing ones, restrict liquidity state transitions, cap total pool size.
Token Minting	`mint_lm_tokens_from_bucket`, `mint_staked_lm_tokens_from_bucket`, `mint_all_lm_tokens`	LM token supply, bucket balances	Can mint governance/reward tokens. Bounded by bucket allocation system but admin controls bucket parameters at init.
Staking Rewards	`set_staking_lm_emission_potentiometer`	LM emission rate (BPS)	Can adjust reward emission rate for LM stakers.
Vesting	`add_vest`, `cancel_vest`	Vesting schedules, beneficiary allocations	Can create and unilaterally cancel vesting schedules. Cancelled tokens return to bucket.
Emergency Patches	`patch_custody_locked_amount`, `patch_custodies_oracles`	Custody locked amounts, oracle feed assignments	Emergency overrides with no audit trail. Can directly set custody locked amounts, bypassing normal accounting.
Irreversible Actions	`disable_tokens_freeze_capabilities`	Token freeze authority	Permanently removes ability to freeze LP/LM tokens. One-way operation.

APPENDIX G: Permissionless Attack Surface

PERMISSIONLESS ATTACK SURFACE

Entries marked with a finding ID are detailed in the main vulnerability section. This appendix provides the complete permissionless surface in one view for operational monitoring and incident response.

Instruction	Constraint	State Modified	MEV / Grief Relevance
`liquidate_long` / `liquidate_short`	Position must exceed `max_leverage`	Closes position, frees custody OI and locked amount, transfers remaining collateral	Liquidator receives liquidation fee. No time-lock gate — liquidation bypasses `min_position_open_time`. MEV opportunity via Jito bundles.
`close_position_long` / `close_position_short` (SL/TP)	Stop-loss or take-profit price reached; caller != owner	Closes position, distributes fees	Executor receives `AUTOMATION_EXECUTION_FEE` (lamports). Partial close forbidden for permissionless SL/TP triggers.
`execute_limit_order_long` / `execute_limit_order_short`	Limit price reached, oracle prices fresh	Opens position from escrowed collateral	Executor receives automation fee + gas reimbursement. Potential for selective execution (only profitable orders).
`distribute_fees` (L4, L8)	Any signer	Converts fee debt to token transfers across LP, LM, protocol, manager, referrer vaults. Zeroes all debt buckets.	Prepended to liquidity/position instructions. Permissionless caller can trigger distribution timing, front-running fee config changes.
`resolve_staking_round`	Any signer; current round must be finalizable	Mints staking rewards, transitions to next round	Permissionless round finalization. Caller could time round transitions to maximize/minimize specific stakers' rewards.
`resolve_position_borrow_fees` (L5)	Any signer; targets any open position	Accrues borrow interest on target position, increases `paid_interest_usd`	Can force-accrue borrow fees on any position. Could be used to push a position closer to liquidation threshold by increasing its fee burden.
`add_liquidity` / `remove_liquidity` (L11)	Any signer (own funds)	Mints/burns LP tokens, updates custody assets and pool AUM	No LP hold period. Sandwich attack around liquidation events is mechanically possible but currently unprofitable (~28 bps round-trip + mark-to-market AUM).
`swap`	Any signer (own funds); pool must allow swap	Transfers tokens between custodies, updates custody assets and collected fees	Swap fees provide MEV resistance. `set_pool_whitelisted_swapper` can restrict to specific callers.
`update_pool_aum`	Any signer	Recalculates `pool.aum_usd` and LP token price	Can force AUM recalculation at advantageous moments (e.g., right before/after large price moves). Affects LP token mint/burn ratios.
`autonom_market_opening` (M5)	Any signer; must be within market open window	Settles overnight funding, updates market state	First caller after market open triggers funding settlement. Timing advantage for positions with favorable funding direction.
`genesis_stake_patch` (L10)	Any signer	Lists as mutable: staking, user_staking, cortex, staking vaults, lm_token_mint PDAs	Deprecated no-op that returns `Ok(())`. Griefing vector via Solana write-lock contention on listed PDAs. See findings.

APPENDIX H: CPI Surface Analysis

APPENDIX H: CPI Surface Analysis

```
FINDINGS
═══════════════════════════════════════════════════════════════════════

[P1] No re-entrancy via self-CPI
     Borrows dropped before CPI; inner targets re-validate all accounts.

[P2] No re-entrancy via external CPI
     All external targets (SPL Token, System, Metaplex) never callback.

[P3] No stale state after CPI
     Accounts .reload()'d before reading post-CPI balances.

[P4] UncheckedAccount properly guarded
     Used only in composites; CPI target enforces PDA init/seeds.

[P5] Consistent PDA signing authority
     Single transfer_authority PDA signs all 32 invoke_signed calls.

[P6] Dead code has zero exposure
     SPL Governance adapter: defined but never called. Recommend removal.
═══════════════════════════════════════════════════════════════════════
```

APPENDIX I: Oracle Surface and Compound Attack Chains

Status:	Resolved
Staleness Laundering and Consensus Degradation have been resolved (see full report body for details). Compound attack left for references purposes	

Three compound attacks, utilizing multiple vulnerabilities on the Oracle surface exist and compound each other

Chain 1: Staleness Laundering, Consensus Degradation, Position Exploitation:

1. Network congestion causes backup oracle cranks to miss their 7s staleness window
2. min_agree=1 pool falls back to trusting primary alone (H2, line 651: return Ok(p))
3. Attacker replays a ~20s stale Switchboard quote via update_oracle — the oracle program accepts it (within 50 slots), and Adrena stores it with timestamp = current_time
4. The laundered timestamp passes the 7s usage staleness check
5. The primary oracle now reports a stale-but-"fresh" price with no backup to contradict it
6. Attacker opens/closes a position against the stale price for profit

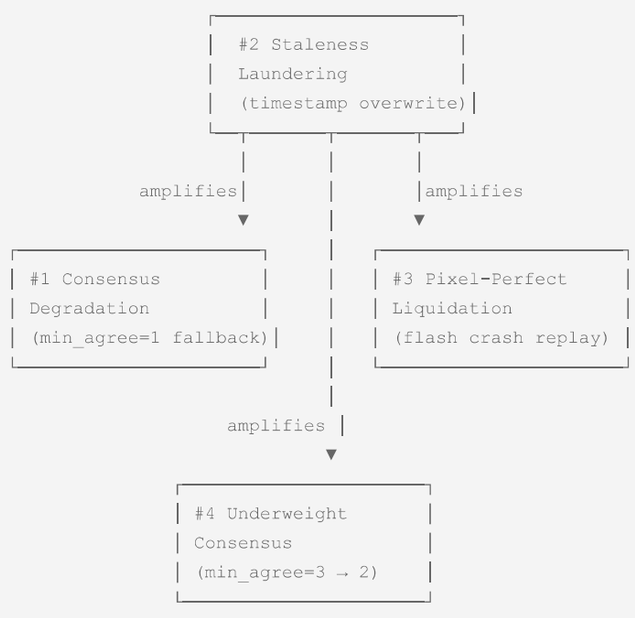
Chain 2: Staleness Laundering, Pixel-Perfect Liquidation:

1. Flash crash occurs — attacker captures the crash-price Switchboard quote
2. Market recovers within seconds
3. Attacker replays the crash-price quote via update_oracle — timestamp is overwritten to current_time
4. The laundered timestamp extends the liquidation window beyond the natural 7s staleness limit
5. The primary oracle now reports a stale-but-"fresh" price with no backup to contradict it
6. Attacker liquidates positions that have already recovered, at the crash price

Chain 3: Staleness Laundering, Silent Consensus Downgrade, Pool Drain:

1. Admin sets min_agree=3 expecting full three-oracle consensus
2. min_agree=3 silently behaves as min_agree=2 — the secondary+tertiary fallback fires even when only two oracles agree
3. Attacker controls two Switchboard quote accounts (secondary + tertiary feed slots)
4. Attacker replays two stale-but-laundered prices that agree with each other
5. These bypass the primary oracle entirely and satisfy the "consensus" check
6. If the primary is also stale, no oracle contradicts the fabricated price
7. Attacker drains the pool at an artificial price

Staleness Laundering is the force multiplier and needs to be patched



APPENDIX J: Positive Observations

Synthetic Custodies cannot be substituted for Custodies

Even though synthetic custodies and custodies share a PDA derivation path (see vulnerabilities list in main body), there is no way to substitute one for the other

Every instruction that uses custodies uses Anchor constraints

```
constraint = custody_token_account.key() == custody.load()?.token_account,
constraint = custody_token_account.mint == custody.load()?.mint,
```

For synthetic custodies `(token_account = Pubkey::default(), mint = Pubkey::default())`, these constraints fail at validation.

Liquidity and swap instructions have further `require` checks

Most accounts handle revival attacks in full

Anchor's `close` attribute is used to reassign account to system program and zero lamports on close

- `delete_user_profile`
- `remove_pool`
- `remove_custody`
- `cancel_vest`
- `liquidate_long`
- `liquidate_short`

Notable exception: `close_position_long` and `close_position_short` do not use Anchor's `close` attribute. They drain lamports manually via `Cortex::transfer_sol_from_owned()` and zero fields via `*position = Position::default()`, but the 8-byte Anchor discriminator is preserved. This leaves the account revivable — see M4 for the full zombie PDA exploit chain.

Well architected CPI surface

- `AccountLoader` borrows are dropped before `invoke_signed`. Inner CPI targets re-run full Anchor account validation.
- External CPI targets are well-known programs that do not issue callbacks to the calling program.
- Token accounts are `reload()`'d after CPI before reading post-swap balances (e.g., `collateral_account.reload()`)
- `UncheckedAccount` is properly guarded. CPI target enforces `#[account(init, seeds=[…])]` or existing PDA validation.
- Single `transfer_authority` PDA (`seeds=[b"transfer_authority", bump]`) is the sole signer for all privileged CPI

APPENDIX J: Positive Observations

Open Interest cannot diverge

Open Interest is tracked in two parallel systems: `trade_stats.oi_long_usd / oi_short_usd`, which feeds funding rate calculations, and `long_positions.size_usd / short_positions.size_usd` which feeds OI cap enforcement and AUM/PnL calculations

If these were to diverge, an attacker could simultaneously bypass OI caps, extract excess funding from existing traders, and hide real pool exposure from AUM — leading to silent LP insolvency

The three core protections in play are:

1. Solana transaction atomicity reliance - The values are always set within the same instruction
2. Lack of independent admin setters prevents accidental or malicious one-sided admin override
3. Single source of truth - Both systems ultimately recompute from the same `position` and are not fed by distinct data

Funding rate sniping is not economically viable

The virtual funding rate has no smoothing, velocity limits, or TWAP. It is recalculated from scratch on every transaction that touches a position (`update_virtual_funding_rate` in `custody.rs:626`). This means an attacker could instantly spike `current_rate_long_to_short` to `max_hourly_funding_rate` by opening a large majority-side position, then collapse it by closing.

At the pure funding level, a mathematical micro-profit is possible when an attacker holds a minority-side position much larger than their temporary majority-side position. The rate increase from the small majority addition benefits the large minority position more than the full rate costs the small majority position. The profitability condition is: `dR/R > M / (m - M)` where `m` is the minority size, `M` is the majority size, and `dR` is the rate change from adding `M`.

This is not economically exploitable:

1. **Per-dollar symmetry blocks self-funding**. An attacker with more USD on the majority side than minority side always pays net funding. The majority position must be larger than the minority to push the rate, so the attacker is always a net payer in this configuration (S1, proven exactly with zero tolerance).
2. **Fee floor dominates the micro-profit**. The 16 BPS `close_position` fee on the temporary majority position exceeds the funding micro-profit in all tested scenarios. To break even on fees with a $10M minority position and $10K manipulation position at 1%/hr max rate requires ~12 minutes of holding, yielding ~$1,900/day, representing a 0.019% daily return on $10M of directional risk (S2, S3).
3. **The minority position is the real exposure**. Any attacker positioned to extract meaningful funding profit (>$1K/day) already holds $10M+ in directional risk on the minority side, where a 1% adverse price move costs $100K. The manipulation adds <0.2% to the funding they would earn anyway simply by being a minority-side participant.

APPENDIX J: Positive Observations

OI arithmetic overflow/underflow is unreachable

Open Interest mutations use bare `+=` and `-=` on `trade_stats.oi_long_usd` and `accounting.size_usd` rather than `checked_add` / `checked_sub`. With `overflow-checks = true` in `[profile.release]` (Cargo.toml:7), these panic on overflow/underflow and abort the transaction rather than silently wrapping. This makes the worst case a DoS (transaction failure), not value extraction.

In practice, neither direction is reachable:

1. Overflow: `max_cumulative_long_position_size_usd` cap enforcement in `update_accounting_after_open_position_long` (custody.rs:1042) rejects positions before OI can approach `u64::MAX` (~$18.4 trillion at 6 decimals). Even with caps disabled, the collateral required is not practically achievable.
2. Underflow: Every `size_usd` subtracted on close/liquidation was previously added on open/increase within the same synchronized tracking system. The only path to `position.size_usd > oi_long_usd` would require a prior operation that decremented OI without closing its corresponding position, which cannot occur (see: OI cannot diverge).
3. Partial close rounding: `downscale` computes `closed = field - floor(field * keep / denom)`, and `subtract` computes `remaining = field - closed`. Both portions are subtracted from OI in sequence, and `closed + remaining = original` exactly.

CLI key handling is sound

The admin CLI (`cli/src/client.ts`, `cli/src/cli.ts`) handles private key material correctly

1. There is no key logging, with all implementations reading from file, constructing `Keypair.fromSecretKey()`, and never passing the result to `console.log`
2. Only public keys are logged, always via `.publicKey.toBase58()`
3. External API calls (price feeds) send only API key headers, never keypair data. The only network destination for signing material is the Solana RPC endpoint via Anchor's transaction submission
4. No private keys or seed phrases appear in source. The only hardcoded cryptographic values are well-known Solana program addresses (SPL Governance, Metaplex Token Metadata)
5. Keypairs are loaded into function-scoped variables or the `AdrenaClient` instance, not persisted to disk, global state, or environment variables
6. The `ANCHOR_WALLET` environment variable stores only the file path, not key material

CLI PDA derivations match on-chain seeds

All PDA derivations in `client.ts` were verified against their corresponding Rust account constraints

No seed mismatches, incorrect byte encodings, or account confusion was identified. The CLI does not implement its own access control. Authorization is delegated to on-chain Anchor `has_one` and `seeds` constraints.

APPENDIX J: Positive Observations

No mainnet deployment automation in repository

The repository contains deployment and initialization scripts exclusively for devnet and localnet environments (`devnet_deploy.sh`, `devnet_full_init.sh`, `devnet_migration.sh`, `local_setup_release38.sh`, `local_setup_release39.sh`). No mainnet deployment script exists in the codebase.

Since source control contains no mainnet automation, accidental mainnet corruption or unauthorized operator deployments are dramatically less likely.

Permissionless instructions correctly separate caller from beneficiary

The protocol exposes 14 permissionless instructions callable by any signer. None allow the caller to redirect funds or corrupt state to their benefit.

The design pattern is consistent: permissionless callers can trigger execution, but all value flows are determined by PDA-derived accounts and on-chain state, never by caller-supplied destinations:

Admin constraint enforcement is consistent across all privileged instructions

Every admin instruction follows the same authorization pattern:

1. `has_one = admin` on the Cortex account, linking the signer to the on-chain admin pubkey
2. `constraint = cortex.load()?.is_initialized()` preventing calls against uninitialized state
3. Pool identity verified via PDA seeds `["pool", pool_name_bytes]`
4. Custody identity verified via `validate_pda(&custody.key(), &pool.key(), &crate::ID)`

No admin instruction was found where the admin check is missing, commented out, or bypassable by an external caller. There is one instance where `has_one = admin` is commented out (`mint_lm_tokens_from_bucket.rs:33`), but an equivalent runtime check at line 105–109 maintains correct gating.

PDA bump validation is complete across all seeded accounts

Every account constraint specifying `seeds = [...]` across all instruction files includes a corresponding `bump` constraint. No missing bumps were found.

Stored bumps are written once at account creation and are immutable thereafter. Since PDA derivation is deterministic, the bump cannot go stale or be manipulated post-initialization.

AUM calculation uses conservative price bands to protect the pool

`get_assets_under_management_usd` (pool.rs:1518) consistently applies oracle confidence bands in the direction that understates pool value

Additionally, `pool_amount_usd.saturating_sub(pool_value_sub_usd)` at line 1923 prevents AUM from underflowing to a wrapped value even under extreme fee debt or PnL scenarios. AUM floors at zero rather than wrapping to `u128::MAX`.

APPENDIX J: Positive Observations

remaining_accounts cannot be manipulated to distort AUM

Several critical instructions pass `ctx.remaining_accounts` to `Pool::get_assets_under_management_usd()` for AUM calculation, which determines LP token mint/redemption amounts. Despite remaining accounts being caller-controlled, the validation is sound.

Every custody account loaded from remaining_accounts is validated against the pool's stored custody pubkeys via positional matching:

```
for (idx, &custody_pubkey) in custodies.iter().enumerate() {
    require_keys_eq!(accounts[idx].key(), custody_pubkey);

    let data = accounts[idx].try_borrow_data()?;
    require!(data.len() >= Custody::LEN, AdrenaError::InvalidCustodyAccount);
    require!(data[..8] == Custody::discriminator(), AdrenaError::InvalidCustodyAccount);
}
```

Because Solana pubkeys are globally unique, `require_keys_eq!` against the pool's stored custody array is a complete validation

There is no way to forge an account with a matching pubkey but different data. Reordered, duplicated, substituted, or random accounts all fail.

Cross-pool confusion is also impossible: custody PDAs are derived from `["custody", pool_key, mint]`, so a custody from pool A can never match pool B's stored pubkey.

Minor observation (informational): the regular custodies loop lacks an explicit `accounts.len() >= custodies.len()` check (only the synthetic custodies loop has one). Too-few accounts cause a Rust index-out-of-bounds panic rather than a graceful `AdrenaError::InvalidCustodyAccount`. The transaction reverts either way without state corruptions.

Swap instruction has layered access control and conservative pricing

The swap instruction implements three independent layers that prevent unauthorized or manipulated token exchange

APPENDIX J: Positive Observations

Stale borrow rate from feeless swap is not economically exploitable

Internal feeless swaps (triggered by `open_or_increase_position_with_swap`) skip `update_borrow_rate`, creating a transient window where the borrow rate reflects pre-swap utilization. Extraction is economically non-viable:

- Each swap - lock-in cycle requires opening a position (16 BPS fee). 100 cycles cost $1,600 to inflict $6.40 excess interest
- $5,120 total excess interest on $8M borrows. Attacker's pro-rata LP share: $20.48 (on $100K deposit in $25M pool) against $216 in round-trip fees
- The two findings are additive, not multiplicative. `checked_ceil_div` rounding adds exactly 0 or 1 per interval regardless of rate magnitude

Compound MEV extraction via permissionless fee operations is not profitable

Three permissionless instructions - `resolve_position_borrow_fees`, `distribute_fees`, and `remove_liquidity` can be sequenced by any wallet in a single transaction. An attacker holding LP tokens could theoretically:

1. Deposit as LP at current AUM
2. Call `resolve_position_borrow_fees` to crystallize borrow interest into fee_debt
3. Call `distribute_fees` to zero fee_debt and transfer tokens to recipients
4. Redeem LP tokens at the post-distribution AUM

`distribute_fees` removes tokens from `staking_reward_token_custody.assets.owned` **proportional to the fee_debt it zeros**. The AUM deduction (fee_debt subtraction) and the token outflow (custody depletion) are economically equivalent and there is no AUM reflation for the attacker to capture.

Both `add_liquidity` and `remove_liquidity` read the same AUM, so the attacker buys and sells at the same price (less round-trip fees)

This result also closes the interaction with the LP sandwich finding (L11) and fee-debt sandwich (state machine I24): the permissionless fee crystallization path does not create an additional MEV vector beyond what those findings already characterize as unprofitable.

Token-2022 transfer hooks and extensions are structurally unreachable

Token-2022 tokens with transfer hooks, transfer fees, or confidential transfer extensions cannot be used as custody tokens.

Even a compromised admin cannot introduce Token-2022 custodies.

APPENDIX J: Positive Observations

Access control mechanism is consistent and centralized

All Admin Instructions correctly reject non Admin users, using a standardized `has_one = admin`

On-Chain Binary Verified Against Source

The deployed mainnet is a bit-exact match against the audited source code.

Program ID:	`13gDzEXCdocbj8iAiqrScGo47NiSuYENGsRqi3SEAwet`
Deploy slot:	`402672276` (2026-02-25T17:44:54Z)
Source branch:	`release/37_4` at commit `5242fcdb`
Build environment:	`solanafoundation/solana-verifiable-build` (Solana v1.18.22, platform-tools v1.41, rustc 1.75.0-dev)
Program content:	3,031,072 bytes — **zero differing bytes**
SHA-256:	`ce9707cbb1156584a953bb831a51aa97c048b34515c2bf9f579b2bc6fa96df6f`

Source Code Free of Trojan Source and Homoglyph Attacks

A full scan of `programs/adrena/src/` confirmed zero non-ASCII characters anywhere in the program source. Specifically, no instances of:

- Bidirectional override characters (`U+202A`–`U+202E`) that reorder displayed code to hide malicious logic from human reviewers
- Zero-width characters (`U+200B`, `U+FEFF`, `U+200C`, `U+200D`) that create visually identical but semantically distinct identifiers — e.g., a PDA seed containing an invisible character would derive a different address than what a reviewer reads
- Homoglyph substitutions (e.g., Cyrillic `а` vs Latin `a`) in identifiers, string literals, or account seeds

These attack classes (collectively known as "Trojan Source", Boucher & Anderson 2021) bypass human code review while remaining visible to compilers. The codebase is clean. See **I17** for recommendations on maintaining this property via CI enforcement.

Protocol has strong public management

The team is publicly doxxed, has significant relevant experience (Wintermute), and is backed by a well-respected public investor (Techstars)

APPENDIX J: Positive Observations

Paths Proven Safe with Formal Analysis

The following properties held in the bounded formal model. These are not vulnerability findings but positive assurances from the formal verification effort.

Oracle selection freshness guard

- `oracle_selection_never_returns_stale_provider`: successful
- Meaning: the reduced oracle-selection model did not return stale providers.

Borrow cadence boundedness

- `borrow_interest_schedule_divergence_exists`: successful
- `borrow_interest_schedule_bounded_divergence`: successful
- Meaning: cadence dependence exists, and the modeled divergence stayed within the asserted bound for the chosen search space.

Funding cadence boundedness

- `funding_schedule_divergence_exists`: successful
- `funding_schedule_bounded_divergence`: successful
- Meaning: cadence dependence exists, and the modeled divergence stayed within the asserted bound for the chosen search space.

Protective-band pricing follow-ups

- `protective_band_liquidation_price_monotonicity`: successful
- `protective_band_exit_numbers_consistency`: successful
- `protective_band_leverage_monotonicity`: successful
- Meaning: these bounded follow-up properties held in the reduced production-replica model.

Fee split conservation

- `fee_debt_nowrap_split_conservation`: successful
- Meaning: the fixed-config fee-split conservation regression held in the current reduced model.

Collateral PnL repricing

- `collateral_pnl_matches_repricing`: successful
- Meaning: the collateral repricing term correctly reports magnitude and direction for all inputs in the bounded domain.

Swap fee boundedness

- `swap_fees_bounded`: successful
- Meaning: in-fees are bounded by amount_in, out-fees by amount_out, and user_receives ≤ amount_out for all inputs in the bounded domain.

AUM PnL direction

- `aum_pnl_direction_correctness`: successful
- Meaning: trader losses increase pool AUM contribution and trader profits decrease it — no sign inversion possible in the bounded domain.

APPENDIX K: Dependency Analysis

Open CVEs

Crate	Version	RustSec	Prod/Dev	Details	Impact
borsh	0.9.3	RUSTSEC-2023-0033	Prod	Unsound parsing of ZST types. Adrena uses borsh for all account serialization via Anchor. Adrena's account types are all sized structs (Position,Custody, Pool, etc.), not ZSTs. The unsoundness requires a type that is !Copy + !Clone + ZST, which none of Adrena's types are.	Not directly exploitable
borsh	0.10.3	RUSTSEC-2023-0033	Prod	Unsound parsing of ZST types. Adrena uses borsh for all account serialization via Anchor. Adrena's account types are all sized structs (Position,Custody, Pool, etc.), not ZSTs. The unsoundness requires a type that is !Copy + !Clone + ZST, which none of Adrena's types are.	Not directly exploitable
curve25519-dalek	3.2.1	RUSTSEC-2024-0344	Prod	Timing side-channel in scalar subtraction. Pulled in by solana-program via solana-zk-token-sdk. Adrena does not perform any ed25519/curve25519 operations itself. The Solana runtime uses it for signature verification, but that happens in the validator, not in the program	Not directly exploitable
ed25519-dalek	1.0.1	RUSTSEC-2022-0093	Prod	Double public key signing oracle attack. Also via solana-program transitively. Adrena never calls ed25519 signing functions. Signature verification is done by the Solana runtime's Ed25519SigVerify precompile, which uses a separate implementation.	Not directly exploitable
keccak	0.1.5	RUSTSEC-2026-0012	Prod	Unsound ARMv8 assembly backend. Pulled in by curve25519-dalek. Solana programs run in the BPF/SBF VM which does not use native ARM assembly. Only affects native execution context	Not directly exploitable
ouroboros	0.15.6	RUSTSEC-2023-0042	Prod	Unsound self-referential struct generation. Used by Anchor internally. Anchor's usage is limited and well-tested	No known Anchor exploit paths exist
bytes	1.7.1	RUSTSEC-2026-0007	Dev	Integer overflow in BytesMut::reserve	Test-only (via tungstenite → solana-pubsub-client)
crossbeam-channel	0.5.13	RUSTSEC-2025-0024	Dev	Double free on Drop	Test-only (via solana-runtime)
quinn-proto	0.10.6	RUSTSEC-2026-0037	Dev	DoS in QUIC endpoints	Test-only (via solana-quic-client)
ring	0.16.20	RUSTSEC-2025-0009	Dev	AES panic with overflow checks	Test-only (via solana TLS stack)
ring	0.17.8	RUSTSEC-2025-0009	Dev	AES panic with overflow checks	Test-only (via solana TLS stack)
time	0.3.36	RUSTSEC-2026-0009	Dev	Stack exhaustion DoS	Test-only
tokio	1.39.3	RUSTSEC-2025-0023	Dev	Broadcast channel !Sync	Test only

A custom vendored fork of the Solana SDK is being used. This fork differs from mainline upstream `v1.18.22` on two commits.

1. `9efdd74b1` is a backported upstream fix. The rbpf crate is the BPF/SBF virtual machine from version 0.8.3 and includes bug fixes for the VM interpreter. This is a positive security change.
2. `c999e0228` is a test only change which works around an Anchor v0.29 lifetime compatibility issue (https://github.com/coral-xyz/anchor/pull/2711). It affects `BanksClient/ProgramTest` harnessing, but does not affect prod as the actual `solana-program` crate is unmodified.

To avoid maintaining a custom fork, and to ensure consistency with modern security updates, we recommend upgrading Solana to 2.x, and Anchor to 0.30 as early as feasible.

Disclaimer

Disclaimer

This report is governed by the Fidesium terms and conditions.

This report does not constitute an endorsement or disapproval of any project or team, nor does it reflect the economic value or potential of any related product or asset. It is not investment advice and should not be used as the basis for investment decisions. Instead, this report provides an assessment intended to improve code quality and mitigate risks inherent in cryptographic tokens and blockchain technology.

Fidesium does not guarantee the absence of bugs or vulnerabilities in the technology assessed, nor does it comment on the business practices, models, or regulatory compliance of its creators. All services, reports, and materials are provided "as is" and "as available," without warranties of any kind, including but not limited to merchantability, fitness for a particular purpose, or non-infringement.

Cryptographic assets and blockchain technologies are novel and carry inherent technical risks, uncertainties, and the possibility of unpredictable outcomes. Assessment results may contain inaccuracies or depend on third-party systems, and reliance on them is solely at the Customer’s risk.

Fidesium assumes no liability for content inaccuracies, personal injuries, property damages, or losses related to the use of its services, reports, or materials. Third-party components are provided "as is," and any warranties are strictly between the Customer and the third-party provider.

These services and materials are intended solely for the Customer’s use and benefit. No third party or their representatives may claim rights to or rely on these services, reports, or materials under any circumstances.