Cyber Apocalypse 2026 — Blockchain — The Far Orchard

Flag:
`HTB{d4m4s_f4r_s34ls_br0k3_b3n34th_g0ld_l34v3s_09bed451e3e5e4f30f091a1c08715795}`

---

1. Challenge overview

> Caldrin follows the gold-flecked trail to the Far Orchard, where Orchard Wardens judge Far-Seals before Crownspire accepts them. […] Prove the rite is broken: wear every approved face, claim all eight seals, and force the Orchard's judgment to collapse in public.

The challenge ships:

- A **halo2 (PLONK / IPA over the Pasta curves)** zero-knowledge circuit written in Rust (`src/circuit.rs`, `src/merkle.rs`, `src/verifier.rs`, `src/main.rs`), plus the full `halo2_proofs` / `halo2_gadgets` / `halo2_poseidon` crates.
- Two Solidity contracts (`setup/src/FarOrchard.sol`, `setup/src/Setup.sol`).
- A live HTTP service (`154.57.164.83:31981`) that deploys a private chain, verifies proofs, and — crucially — **signs an EIP-712 "honor receipt"** whenever a proof verifies.

The rite (intended flow)

The Orchard registry commits to **8 approved public keys** (the "Far-Seals") in a depth-8 Merkle tree. To honor seal `i` you are supposed to prove, in zero knowledge, that:

1. you know the secret key `sk` behind approved public key `pk_i` (`pk_i = [sk]·G`),
2. `pk_i.x` is the leaf at position `i` of the registry Merkle tree, and
3. a **nullifier** `nf = [sk]·G` (its x-coordinate) is exposed so a seal can't be honored twice.

The validator EOA (`validatorSigner`) signs `Honor(claimant, sealId, nullifier)` on a valid proof; the `FarOrchard` ERC-721 mints seal `i` after checking that signature on-chain.

Win condition (`setup/src/Setup.sol`)

```
function isSolved() external view returns (bool) {
    uint256 completeBitmap = (1 << 8) - 1;              // 0xFF
    return orchard.honoredCount() == 8 &&
           orchard.honoredBitmap() == completeBitmap;    // all 8 honored
}
```

We must honor **all 8 seals** — but we don't know a single secret key.

---

2. The vulnerability — an unanchored variable base

The whole rite hinges on one line of the circuit. Look at how the public key is computed in `src/circuit.rs`:

```
// g_d is a *witnessed* point supplied by the prover
let g_d = NonIdentityPoint::new(ecc_chip.clone(), .., self.g_d)?;

let sk_cell = /* witnessed sk */;
let sk_scalar = ScalarVar::from_base(ecc_chip.clone(), .., &sk_cell)?;

// pk = [sk] * g_d          ⟵ variable-base multiply against an ATTACKER poin
let (pk, _) = g_d.mul(.., sk_scalar)?;

// nullifier = [sk] * G     (G = the real generator, fixed base)
let nullifier_base = FixedPointBaseField::from_inner(ecc_chip, FarOrchardBaseFi
let nf = nullifier_base.mul(.., sk_cell.clone())?;

let leaf = pk.extract_p();          // leaf committed to the tree = pk.x
```

And the ECC chip is deliberately constructed in an insecure mode:

```
EccChip::construct(self.ecc_config.clone(),
    halo2_gadgets::ecc::chip::CircuitVersion::InsecureUnanchoredBase)
```

The public inputs are `[merkle_root, nullifier_x, leaf_pk_x]`.

The base point `g_d` is fully controlled by the prover and is never constrained to be the generator `G`. The honest circuit is *supposed* to force `pk = [sk]·G` (so the leaf really is the public key of a secret you know). Instead it computes `pk = [sk]·g_d` for an arbitrary attacker-chosen `g_d`. That completely decouples the leaf from any secret:

> For **any** target point `P` (i.e. any leaf `pk_i` in the registry), we can make the circuit output `leaf = P.x` **without knowing the discrete log of `P`** — just pick any `sk` and set `g_d = [sk⁻¹]·P`, so that `[sk]·g_d = P`.

Meanwhile the nullifier `nf = [sk]·G` depends only on our chosen `sk`, so we can hand each of the 8 seals a **different `sk`** and thus a different, fresh nullifier (required by the on-chain `nullifierUsed` mapping).

This is the "judgment without an anchor" from the intro text: the variable base is *unanchored*, so the ZK proof no longer proves possession of an approved key — it only proves that we picked a point whose x-coordinate happens to equal a registry leaf, which anyone can do.

Forgery recipe (per seal `i`)

Given the registry leaf `pk_x_i` (from `GET /api/info`):

1. Recover the curve point `P_i` from its x-coordinate: `y² = x³ + 5` on Pallas, pick either root.
2. Choose a small distinct secret `sk_i` (we use `sk_i = i + 2`).
3. Compute `g_d_i = [sk_i⁻¹]·P_i` (inverse taken in the **scalar** field `Fq`).
4. Witness `sk = sk_i`, `g_d = g_d_i`, and the genuine Merkle authentication path for leaf `i` (we know all 8 leaves, so we rebuild the tree with the challenge's own `merkle_crh`).
5. Public inputs become `[official_root, ([sk_i]·G).x, pk_x_i]`.

Then `[sk_i]·g_d_i = P_i ⇒ leaf = pk_x_i` (matches seal `i`), the Merkle path yields the official root, and the nullifier `([sk_i]·G).x` is unique per seal. A perfectly valid proof for a seal we have no key for.

---

3. The service protocol

`/docs` documents the HTTP API (all under `/api`, mirrored under `/rpc`):

Endpoint	Purpose
`POST /api/launch`	Deploy a fresh private chain → `RPC_URL`, `PRIVKEY`, `SETUP_CONTRACT_ADDR`, `WALLET_ADDR`, `uuid`.
`GET /api/info`	Merkle root, the 8 leaves (`pk_x`, `seal_id`), orchard/setup addresses, `validator_signer`.
`POST /api/verify`	Submit `{proof, seal_id}`; on success returns a **signed honor receipt** `{claimant, seal_id, nullifier, signature}`.
`GET /api/flag`	Returns the flag once `isSolved()` is true.

Two operational details that matter:

- **`/api/verify` is bound to the session cookie set by `/api/launch`.** Launch, verify, and the on-chain calls must all happen in one HTTP session (`requests.Session`).
- The Merkle tree is **deterministic** across launches (same 8 leaves / root every time), so the 8 forged proofs can be generated once and reused.

The proof wire format (`src/merkle.rs::serialize_proof`) is: `hex( root[32] || nullifier[32] || leaf[32] || proof_len_le[4] || proof_bytes )`.

---

4. Exploitation

4.1 Proof generator (Rust)

The forger reuses the challenge's own `circuit` / `merkle` modules, generates keys with the same parameters as the validator (`k = 13`), forges a proof per seal, and locally re-verifies each one before writing it out. Full source: `src/bin/solve.rs` (see repo). Core loop:

```
let target = recover_point(pk_x);            // P_i from x-coordinate
let sk_int = (index as u64) + 2;             // distinct small secret
let sk_base = pallas::Base::from(sk_int);
let inv = pallas::Scalar::from(sk_int).invert().unwrap();
let g_d = (pallas::Point::from(target) * inv).to_affine();   // [sk^-1]·P_i

let nullifier_x = merkle::derive_nullifier(sk_base);         // ([sk]·G).x
let circuit = FarOrchardCircuit {
```

```
    sk: Value::known(sk_base),
    g_d: Value::known(g_d),
    merkle_path: Value::known(path_arr),      // real path from rebuilt tree
    merkle_pos: Value::known(index as u32),
};
let public_inputs = [expected_root, nullifier_x, pk_x];
// create_proof(...) → serialize_proof(...) → /tmp/proof_i.hex
```

Build & run:

```
cargo build --release --bin solve
./target/release/solve        # reads /tmp/tree.json, writes /tmp/proof_{0..7}.
```

Output confirms the rebuilt root equals the official root and all 8 proofs verify locally.

4.2 Driver (Python)

One session: launch → info → (reuse proofs) → verify all 8 → `honorSeal` on-chain → flag. Full source: `writeups/exploit/combined.py`. Key parts:

```
launch = s.post(BASE + "/api/launch").json()
info   = s.get(BASE + "/api/info").json()

for i in range(8):                        # collect 8 signed receipts
    proof = open(f"/tmp/proof_{i}.hex").read().strip()
    r = s.post(BASE + "/api/verify", json={"proof": proof, "seal_id": i}).json(
    receipts.append(r)                    # {claimant, seal_id, nullifier, signa

for r in receipts:                        # honor each seal from the claimant wa
    tx = orchard.functions.honorSeal(
        r["seal_id"],
        bytes.fromhex(r["nullifier"][2:]),
        bytes.fromhex(r["signature"]),
    ).build_transaction({...})
    ... send & wait ...

print(s.get(BASE + "/api/flag").json())
```

4.3 Result

```
launched wallet 0x04824F50D3f15d24Df61e6623e85a91F9F56617e
receipt seal 0..7: ok
honorSeal(0..7) status=1
honoredBitmap = 0b11111111
GET /api/flag: {"flag": "HTB{d4m4s_f4r_s34ls_br0k3_b3n34th_g0ld_l34v3s_09bed451
```

All eight Far-Seals honored; the rite collapses.

Flag:
`HTB{d4m4s_f4r_s34ls_br0k3_b3n34th_g0ld_l34v3s_09bed451e3e5e4f30f091a1c08715795}`

---

5. Root cause & fix

- **Root cause:** `pk` is derived by a *variable-base* scalar multiplication against a prover-witnessed point `g_d`, and the ECC gadget runs in `InsecureUnanchoredBase` mode. The circuit never binds `g_d` to the canonical generator `G`, so proving "the leaf is my public key" degenerates to "the leaf is some point's x-coordinate" — trivially satisfiable for every registry leaf without any secret.
- **Fix:** derive the public key with a **fixed-base** multiply against the real generator (`pk = [sk]·G`, the same fixed base used for the nullifier), or otherwise constrain `g_d == G` in-circuit, and use the secure (anchored) ECC circuit version. With `g_d` fixed, `leaf = ([sk]·G).x` can only be produced by someone who actually knows `sk`, restoring soundness of the rite.