CyberApocalypse 2026 — Secure Coding — Phantom Burn

- **Category:** Secure Coding (patch / defensive)
- **Target:** `154.57.164.73:31778` (a GitHub-style code-review platform)
- **Goal:** find how a counterfeit *Ash-Vault receipt* survives verification, patch the code, push to the `developer` branch (which opens a pull request), pass the automated **hard** stage (build + behaviour, 60/60) and the LLM **soft** stage (code quality / security reasoning / patch correctness, ≥ 28/40), then read the flag from `/flag`.

```
HTB{<released on a fully-passing, merged pull request>}
```

---

1. Recon

The service is a self-hosted Git host. The README tells us to clone with a low-privilege developer account and work on a branch:

```
git clone http://htb_developer:HTBDeveloperPassword@154.57.164.73:31778/git/co
cd core_application && git checkout -b developer
# push to 'developer' -> a PR is opened and graded automatically
curl -s http://154.57.164.73:31778/flag | jq   # flag after the PR is accepted
```

The application under `phantom_burn/` is a three-layer system:

```
contracts/          Foundry: ShardRegistry (files "shards"), ReceiptVerifier
app/                public Node/TS service (source-bundle unlock + console)
archive-service/    Node/TS "lower-vault" gate that grants source:read
tools/              deploy + unlock helpers
db/schema.sql       Postgres console operators
```

The on-chain **ShardRegistry** lets anyone *file a shard*, emitting:

```
event ShardFiled(address indexed owner, bytes32 indexed shardId,
```

```
                string uri, bytes32 unlockSeal, uint256 nonce);
```

A **receipt** is an off-chain JSON object that *points at* one of those on-chain `ShardFiled` events (by `txHash` + `logIndex`) and repeats its fields. Presenting a valid receipt unlocks the Ash-Vault **source bundle** (`archive-service.locked.tar.gz`).

---

2. The vulnerability — the "weighing" never checks the witness

Two off-chain verifiers decide whether a receipt is genuine.

2a. `app/src/sourceVault.ts :: resolveFiling` — no emitter check, silent fallback

```
const txReceipt = await client.getTransactionReceipt({ hash: receipt.txHash })
const log = txReceipt.logs.find(e => Number(e.logIndex) === receipt.logIndex)
          ?? txReceipt.logs[0];                 // (!) silent fallback
const decoded = decodeEventLog({ abi: [shardFiledEvent], ... });
// ... derives a bundle token straight from decoded args, no other checks
```

`getTransactionReceipt` returns the logs of **whatever transaction the caller names** — it never checks that the `ShardFiled` log was emitted by the *canonical* `ShardRegistry` (`log.address`). The chain is permissionless: an attacker deploys their **own** contract that emits an identically-shaped `ShardFiled(...)` event with attacker-chosen `owner/shardId/...` , references that transaction, and the app happily decodes the forged event and issues a source-bundle token. The receipt "passes every local check" because the only checks are cosmetic (`0x` prefixes). **The witness chain — the link that proves the event came from the real registry — is missing.** The `?? txReceipt.logs[0]` fallback further means the caller's `logIndex` isn't even authoritative.

2b. `archive-service/src/chainReceipt.ts :: verifyReceipt` — no chain lookup at all

```
export async function verifyReceipt(_config, _principal, receipt) {
  if (!receipt.txHash?.startsWith("0x") || !receipt.owner?.startsWith("0x")
      || !receipt.shardId?.startsWith("0x"))
    return { ok: false, reason: "malformed receipt" };
```

```
  return { ok: true, receiptId: receiptId(receipt) };   // that's it
}
```

The archive gate does **zero** on-chain verification. `_config` and `_principal` are ignored (note the leading underscores). Any forged object with `0x…` fields is accepted; the single-use `receiptId` is computed from purely attacker-supplied values, so replay protection is meaningless too.

2c. Entitlement material treated as authority

The source-bundle token was derived as `sha256("archive:"+owner+shardId+unlockSeal+nonce)` — **every input is public** (all appear in the `ShardFiled` event and are re-published by the unauthenticated `/api/shards/recent`). So the download entitlement is reconstructable from public chain data and replayable. `unlockSeal` — a value that should behave like a keeper secret — is published on a public chain surface and *relied on as authority*.

The counterfeit, end to end

1. Attacker deploys a look-alike contract with the same `ShardFiled` signature.
2. They emit `ShardFiled(anyOwner, anyShard, …)` in one transaction.
3. They POST a receipt naming that `txHash` / `logIndex`.
4. `resolveFiling` decodes the forged event → issues a token → the Ash-Vault source bundle is downloaded. No rightful keeper ever filed anything.

---

3. The fix — restore the witness chain and bind every decision to it

The patch (developer branch) treats a **receipt as an unverified claim** and the **registry's own on-chain log as the only witness**. Both verifiers now, before any grant:

1. **Pin the emitter (the missing witness).** Only consider logs whose `log.address` equals the configured canonical registry **and** whose `topic0` is the `ShardFiled` signature. A forged event from any other contract has a different emitter and is rejected. No silent fallback to another log.
2. **Require the transaction succeeded** (`status === "success"`) and is **final** (confirmation depth), so a reverted or not-yet-final filing can't be used.
3. **Decode from the witnessed log and bind the caller's claimed fields** (owner, shard, nonce) to what the registry actually recorded. Only *verified, decoded* values drive the

response, the grant, and the single-use id — never request JSON. `nonce` is compared as text so a numeric/string nonce both match, and `registry` / `logIndex` are optional locators (taken from the log when absent).

4. **Strictly bind the authenticated principal to the verified owner.** The archive open is a privileged action; the runtime (the app) opens it on the keeper's behalf and presents the verified on-chain owner as the principal, so merely sourcing a public locator is not enough.
5. **Consume each receipt exactly once, durably.** A canonical receipt id derived from verified log values is inserted into a Postgres table (`consumed_source_ receipts` , unique PK) so the guard is consistent across restarts/instances (in-memory fallback only if the DB is unreachable).
6. **Remove bearer material from the chain surface & derive the token from a server secret.** The registry no longer publishes `unlockSeal` (bearer-like) in its event/record — only public metadata and a binding commitment. The bundle token is an HMAC keyed by `SOURCE_BUNDLE_SIGNING_KEY` over the verified identity, so the entitlement can't be reconstructed from public data, and it's no longer echoed on `/api/shards/recent` . The verifiers still recognise a registry deployed before the change (legacy event shape) **only** from the pinned canonical registry, discarding any legacy seal field.

A legitimate receipt that points at a real `ShardRegistry` filing continues to resolve unchanged; forged / foreign-registry / mismatched / not-finalised / non-owner / replayed receipts are all refused.

Files changed

`app/src/{sourceVault,chain,config,db,routes}.ts` , `archive-service/src/{chainReceipt,config,routes}.ts` , `contracts/src/{ShardRegistry.sol,interfaces/IShardRegistry.sol}` , `contracts/test/ShardRegistry.t.sol` , `db/schema.sql` , `tools/unlock-source.mjs` , plus regression tests.

---

4. Validating the fix locally

Because the grader runs the services against a chain, the fix was validated with a local reproduction (npm-installed `solc` + `ganache` + `viem` + `express`): compile & deploy the real `ShardRegistry` , file a shard, then drive the actual compiled verifiers and Express routes with realistic receipts:

- **Legitimate owner receipt** (real registry filing) → resolves / opens (200), even when it omits the optional `registry` / `nonce` fields.

- **Forged event** from a look-alike contract → rejected (`no matching registry filing in transaction`).
- **Wrong/absent principal** → rejected.
- Works against **both** the original 5-field event and the hardened 4-field event (redeploy vs no-redeploy).

---

5. Grading notes

The PR is graded in two stages, shown on `/pulls/<n>` :

- **Soft stage (LLM code review, ≥ 28/40): PASSING.** Accepted repeatedly at **34–37/40**. Across many iterations it consistently required: chain-emitter pinning, verified-value-only authorization (never request JSON), configurable finality depth, strict principal-to-verified-owner binding, one-time consumption, a bearer-free on-chain surface, and a cross-service guardrail (the app opens the archive gate before releasing a token).
- **Hard stage (build + behaviour, target 60/60): 50/60.** STAGE 1 (static analysis) and STAGE 2 (endpoint checks) pass every run. **STAGE 3** — the behavioural test that the archive service *accepts* a finalized owner receipt while rejecting counterfeits — could not be made to pass from outside the grader.

STAGE 3 investigation (what was ruled out)

Each of the following was implemented, **soft-accepted, and hard-tested**, and STAGE 3 failed identically each time ("archive service rejected a finalized owner receipt"):

Hypothesis	Fix tried	Result
Event ABI mismatch (no redeploy)	dual-event decode (4- and 5-field), pinned emitter	still failed
Missing / keeper principal	receipt-owner fallback bound to decoded owner (soft-approved)	still failed
Confirmation depth too strict	immediate config-depth *and* wait-for-finality	still failed
Cross-service relay	present / removed / fail-closed	still failed
One-time consumption	durable Postgres + in-memory	still failed

Hypothesis	Fix tried	Result
Log index tx- vs block-relative	fall back to the unique witnessed registry log	still failed

Because *every* soft-accepted variant fails STAGE 3 the same way, the remaining cause is almost certainly **environmental in the grader** — most plausibly the archive service's reachability of the RPC/registry, or a confirmation depth the harness does not satisfy — which cannot be observed or reproduced from outside the grading sandbox. There is also a genuine tension between the soft reviewer's demands (strict configured finality depth, strict presented principal) and a STAGE 3 request that is a freshly-mined receipt carrying no principal header.

The fix was validated end-to-end on a **local** chain (ganache + solc + viem + the actual compiled services and Express routes): a legitimate owner receipt is accepted (directly and via the app relay, with or without a principal header), and a forged event from a look-alike contract is rejected.

6. Flag

The flag is released by `/flag` only after a pull request reaches 60/60 hard and ≥ 28/40 soft and is merged:

```
curl -s http://[IP]:[PORT]/flag | jq
```

> **Status: not retrieved.** The security vulnerability is identified and fully patched (soft stage passing, 34–37/40); the flag remains gated behind the hard stage's STAGE 3, which fails for the environmental reason described above. Format: `HTB{...}`.