Cyber Apocalypse 2026 — Quantum — The Counting House

Category: Quantum **Challenge:** The Counting House **Target:** `154.57.164.69:32674` (HTTP / JSON API, served by gunicorn) **Flag:** `HTB{f0rg3d_r34d_4nd_s3ttl3d_t0_th3_s1lv3r_06ce5ed4c5bb6b35f8a5774905c105cd}`

---

1. Overview

> *Eastreach is where the rich houses go to buy the things that actually matter... The Gilded Knife runs the place and built it so you cannot cheat it. You show a note to get a seat, your bid goes into a sealed book nobody reads, and when it is done you pay the winning price exactly.*

The challenge is a full "auction house" with four gates, each of which is a distinct quantum-information attack:

Stage	Story	Endpoint	Quantum concept	What you must break
1. **Enter**	Fake a bearer note to get a seat	`POST /api/enter`	Stabilizer / CSS subspace states	Prepare a uniform superposition over a linear code you can't be given
2. **Book**	Read the sealed book you're not allowed to read	`POST /api/book`	BB84 conjugate-basis encoding	Recover the basis *and* the bit of every qubit
3. **Seal**	Beat the seal nobody has ever beaten	`POST /api/seal/{commit,peek,open}`	Quantum bit-commitment binding	Break binding with EPR steering, 24 rounds × 8 strands
4. **Settle**	Pay the clearing price and walk out with the deed	`POST /api/settle`	(stage 1 again)	Forge a note whose value equals the exact highest bid

Despite the fancy front-end (`index.html` , `app.js` , `style.css`), everything interesting lives in a small JSON API. The `/api/market` sheet documents the rules:

```
{
  "note": {
    "checks": 4, "entry_value": 4919, "qubits": 8,
    "scheme": "a note of value v is the subspace state |A_v>, A_v = ker(H_v); the
public
               parity-check H_v has 4 rows over 8 bits, row i =
sha256('eastreach-note-v{v}-{i}')
               truncated to 8 bits, skipping rows that do not raise the rank",
    "verify": "lock a note by submitting a circuit on 8 qubits; the house
prepares it and checks
               every Z-basis reading lies in A_v and every X-basis reading lies
in A_v^perp"
  },
  "book": { "bits": 16, "rivals": 6,
            "note": "each rival's sealed bid is encoded qubit by qubit on the
clearing work-tape;
                     you may measure any work-tape qubit you can reach" },
  "settlement": { "rule": "the deed goes to a bidder who passes every round of
the seal-check and
                           then pays the clearing price exactly; the clearing
price is the highest
                           sealed bid in the book",
                  "seal_rounds": 24, "seal_strands": 8, "tries": 6 },
  "gates": "notes: H X Z on a qubit index, CX control,target;
            strands: I X Y Z H S SDG on a|b, CX among a,b"
}
```

Session flow: `POST /api/new` → `{token}`, then every call carries that `token`.

2. Stage 1 — Forging a bearer note (`/api/enter`)

The scheme

A note of value `v` is the **subspace state**

$$|A_v\rangle \;=\; \frac{1}{\sqrt{|A_v|}} \sum_{x \in A_v} |x\rangle, \qquad A_v = \ker(H_v) \subseteq \mathbb{F}_2^8.$$

`H_v` is a public `4 × 8` parity-check matrix. Row `i` is `sha256("eastreach-note-v{v}-{i}")` "truncated to 8 bits", keeping only rows that raise the rank, until 4 independent rows exist. So `A_v` is a `[8,4]` linear code, and `|A_v⟩` is the uniform superposition over its codewords — a **CSS / stabilizer state**.

The house verifies a submitted 8-qubit circuit by:

- measuring in the **Z basis** and checking every outcome `x` satisfies `H_v · x = 0` (i.e. `x ∈ A_v = ker H_v`);
- measuring in the **X basis** and checking every outcome `y` lies in `A_v^⊥`.

This is exactly the signature of the codeword-superposition state: measuring `|A_v⟩` in Z yields uniform elements of `A_v`; applying `H^{⊗8}` (X basis) yields the **dual code** `A_v^⊥ = rowspace(H_v)`. So if we prepare `|A_v⟩` for the *correct* `H_v`, both checks pass with certainty.

Nailing down the byte encoding

The only ambiguity is "sha256 truncated to 8 bits". A quick brute over `{first byte, last byte} × {MSB-first, LSB-first}` and submitting each candidate settled it instantly:

```
byteidx=0  order=msb  -> {'seated': False}
byteidx=0  order=lsb  -> {'seated': False}
byteidx=-1 order=msb  -> {'seated': False}
byteidx=-1 order=lsb  -> {'seated': True}   ✓
```

H_v uses the *last* byte of the digest, bit `k = (byte >> k) & 1` (LSB-first). In Python this is exactly `int.from_bytes(sha256(...).digest(),'big') & 0xFF` read low-bit-first.

Preparing the codeword superposition

Given `H_v`, compute a basis `G` (4×8) of `A_v = ker H_v`, put it in RREF, and encode:

1. Put each **pivot** qubit into `|+⟩` with an `H` gate — these carry the 4 free "message" bits, giving `∑_u |u⟩` over the pivots.
2. For every **non-pivot** column `j`, the codeword bit is `∑_i R[i][j]·u_i`, so apply `CX(pivot_i → j)` for each `R[i][j]=1`.

This produces $\sum_u |u \cdot G\rangle = \sum_{x \in A_v} |x\rangle$ on the 8 physical qubits — no ancillas needed.

For `entry_value = 4919` the parity-check rows (last-byte, LSB) are `[0,1,1,0,0,0,1,0]`, `[0,1,0,1,1,0,1,0]`, `[1,0,0,1,0,1,1,1]`, `[0,1,1,0,1,1,0,0]`, and the note circuit is:

```
[["H",0],["H",1],["H",2],["H",3],
 ["CX",2,4],["CX",3,4],["CX",1,5],["CX",3,5],
 ["CX",1,6],["CX",2,6],["CX",0,7],["CX",2,7]]
```

→ `{"seated": true}`. **Seat taken.**

---

3. Stage 2 — Reading the sealed book (`/api/book`)

Each of the 6 rivals has a 16-bit bid laid out "qubit by qubit". `POST /api/book {token, bidder, position, basis, shots}` measures one work-tape qubit (`bidder 0..5`, `position 0..15`, `basis Z|X`) and returns `shots` outcomes.

Measuring bidder 0 across all positions in both bases reveals the trick — each qubit is one of the four **BB84 states**:

Prepared state	Z-basis reading	X-basis reading
`|0⟩`	deterministic **0**	random
`|1⟩`	deterministic **1**	random
`|+⟩`	random	deterministic **0**
`|−⟩`	random	deterministic **1**

So every bid bit is encoded in a **random conjugate basis** — a "sealed book" you supposedly can't read without knowing the bases.

Why it's readable anyway: each `/api/book` call *re-prepares* the qubit fresh (independent shots, no measurement back-action carried between calls). So for each qubit we simply:

1. Measure in **Z** with many shots. If the outcomes are all identical → it was a Z-basis state; the bit is that value.
2. Otherwise measure in **X** → it was an X-basis state; the bit is that (deterministic) value.

Either way, **bit = the deterministic outcome**. That recovers all 6 × 16 bits, i.e. every sealed bid. The 16 positions are read **MSB-first** (`position 0` = most-significant bit) — see stage 4 for how that was confirmed. The highest of the six values is the **clearing price**.

---

4. Stage 3 — Beating the seal (`/api/seal/*`)

This is the "seal nobody has ever beaten": a **quantum bit-commitment / coin-flip** run for **24 rounds**, each with **8 strands**.

Per round the protocol is:

1. `/api/seal/commit {token, slots}` — you submit 8 "strands", each a **2-qubit circuit over qubits `a` and `b`** (gates `I X Y Z H S SDG` on `a|b`, plus `CX` among `a,b`). The house keeps each `b`, you keep each `a`. It returns a **challenge basis** for the round: `{"challenge": 0|1}` (0 = Z, 1 = X).
2. `/api/seal/peek {token, basis}` — you measure your kept `a`-qubits in a basis of your choice; returns `a_outcomes` (8 bits).

3. `/api/seal/open {token, values}` — the house measures each `b` in the **challenge basis** and checks your `values` match its outcomes.

A classical committer is *bound*: it would have to fix, at commit time, a value that reveals consistently in **both** Z and X — impossible, which is the whole point of a binding commitment.

The break — EPR steering. Make every strand a **Bell pair**:

```
[["H","a"],["CX","a","b"]]     // |Φ+> = (|00> + |11>)/√2
```

`|Φ+⟩` is perfectly correlated in *both* conjugate bases:

$$|\Phi^+\rangle = \tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \tfrac{1}{\sqrt{2}}(|{+}{+}\rangle + |{-}{-}\rangle).$$

The commit fixes nothing — the pair is undetermined until measured. The order of operations lets us **learn the challenge basis before we peek**, so we simply:

- read the round's `challenge`;
- **peek our `a`-qubits in that same basis** → because of the Bell correlation, each `a` outcome equals what the house's `b` will give in that basis;
- **open with `values = a_outcomes`**.

Every strand matches, every round passes:

```
round 0  chal=1 held=True passes=1
round 1  chal=0 held=True passes=2
...
round 23 chal=1 held=True passes=24 sealed=True
```

→ `{"held": true, "passes": 24, "rounds_done": 24, "sealed": true, "total": 24}`. **Seal beaten.** This is textbook: bit-commitment/coin-flipping is impossible to make *both* hiding and binding against a quantum adversary holding half of an entangled pair — the committer defers and steers.

---

5. Stage 4 — Settling with a forged note (`/api/settle`)

`POST /api/settle {token, circuit, value}` grants the deed to a bidder who (a) passed all 24 seal rounds and (b) **pays the clearing price exactly** with a valid note *of that value*.

So `value` must equal the **highest sealed bid** we read in stage 2, and `circuit` is `|A_value⟩` built exactly as in stage 1 but for `v = value` (a forged note whose subspace corresponds to the clearing price).

The one loose end is the 16-bit ordering. With only two candidates (MSB-first vs LSB-first) and 6 settlement tries, we just try both. In the winning run the six bids (MSB-first) were:

```
bidder0=13050  bidder1=5414  bidder2=945  bidder3=8629  bidder4=30704
bidder5=47723
```

- `value = 54877` (LSB interpretation) → `{"settled": false, "reason": "that is not the clearing price"}`
- `value = 47723` (MSB interpretation, the true max) →

```
{"deed":
"HTB{f0rg3d_r34d_4nd_s3ttl3d_t0_th3_s1lv3r_06ce5ed4c5bb6b35f8a5774905c105cd}",
 "settled": true}
```

Position 0 is the most-significant bit. Deed acquired.

---

6. Flag

```
HTB{f0rg3d_r34d_4nd_s3ttl3d_t0_th3_s1lv3r_06ce5ed4c5bb6b35f8a5774905c105cd}
```

---

7. Full exploit

`sol.py` — note forgery + helpers:

```
import hashlib, json, urllib.request
B = "http://154.57.164.69:32674"; V = 4919; N = 8

def post(path, obj):
    req = urllib.request.Request(B + path, data=json.dumps(obj).encode(),
                                 headers={'Content-Type': 'application/json'})
    return json.loads(urllib.request.urlopen(req).read())

def new(): return post("/api/new", {})['token']

def rref(mat):
    mat = [r[:] for r in mat]; r = 0; pivots = []
    for c in range(N):
        piv = next((rr for rr in range(r, len(mat)) if mat[rr][c]), None)
        if piv is None: continue
        mat[r], mat[piv] = mat[piv], mat[r]
        for rr in range(len(mat)):
            if rr != r and mat[rr][c]:
                mat[rr] = [a ^ b for a, b in zip(mat[rr], mat[r])]
        pivots.append(c); r += 1
    return r, mat[:r], pivots

def build_H(v=V):
    H = []; i = 0
    while len(H) < 4 and i < 200:
        b = hashlib.sha256(f'eastreach-note-v{v}-{i}'.encode()).digest()[-1]  #
LAST byte
        bits = [(b >> k) & 1 for k in range(8)]                                #
LSB-first
        if rref(H + [bits])[0] > rref(H)[0]:
            H.append(bits)
        i += 1
    return H

def kernel(H):
    r, R, pivots = rref(H); free = [c for c in range(N) if c not in pivots];
```

```
    basis = []
    for f in free:
        vec = [0]*N; vec[f] = 1
        for ri, pc in enumerate(pivots): vec[pc] = R[ri][f]
        basis.append(vec)
    return basis

def note_circuit(v=V):                      # prepare |A_v> = sum_{x in ker H_v}
|x>
    G = kernel(build_H(v)); r, R, pivots = rref(G)
    ops = [["H", pc] for pc in pivots]
    for j in range(N):
        if j in pivots: continue
        for i, pc in enumerate(pivots):
            if R[i][j]: ops.append(["CX", pc, j])
    return ops
```

`solve.py` — end-to-end:

```
from sol import *

def read_qubit(tok, bidder, pos, shots=60):
    z = post("/api/book", {"token": tok, "bidder": bidder, "position": pos,
"basis": "Z", "shots": shots})['outcomes']
    if len(set(z)) == 1: return z[0]                          # deterministic in
Z  -> Z-state
    x = post("/api/book", {"token": tok, "bidder": bidder, "position": pos,
"basis": "X", "shots": shots})['outcomes']
    return x[0] if len(set(x)) == 1 else round(sum(z)/len(z)) # deterministic in
X  -> X-state

def main():
    tok = new()
    post("/api/enter", {"token": tok, "circuit": note_circuit()})         #
stage 1: seat

    bids = []                                                             #
stage 2: read book
    for b in range(6):
        bits = [read_qubit(tok, b, p) for p in range(16)]
        bids.append(int(''.join(map(str, bits)), 2))                      # MSB-
first
    clearing = max(bids)

    bell = [["H", "a"], ["CX", "a", "b"]]                                  #
stage 3: beat seal
    for _ in range(24):
        c = post("/api/seal/commit", {"token": tok, "slots": [bell]*8})
        basis = "Z" if c['challenge'] == 0 else "X"                        #
peek in the challenge basis
        pk = post("/api/seal/peek", {"token": tok, "basis": basis})
        o = post("/api/seal/open", {"token": tok, "values": pk['a_outcomes']})
        assert o['held']
    assert o['sealed']
```

```
    r = post("/api/settle", {"token": tok, "circuit": note_circuit(clearing),
"value": clearing})
    print(r)                                                                  #
stage 4: deed + flag

main()
```

---

8. Takeaways

- **Notes** are stabilizer/CSS codeword superpositions — verifiable by dual Z/X measurements. Preparing them reduces to linear algebra over $\mathbb{F}_2$ (kernel + RREF + CNOT encoder).
- **The sealed book** is BB84-flavored conjugate-basis hiding, but with re-preparable qubits and unlimited measurements the "no-cloning" protection evaporates: probe both bases, the deterministic one leaks the bit.
- **The unbeatable seal** is quantum bit-commitment, and the classic no-go theorem *is* the exploit: hold half of a Bell pair, learn the challenge basis, and steer — binding is broken with certainty.
- **Settlement** just chains stage 1 to the value recovered in stage 2.