Cyber Apocalypse 2026 — Three Tankards and a Lie

Challenge: Three Tankards and a Lie **Category:** Coding (filed under Forensic per naming convention) **Target:** `154.57.164.72:30699` **Flag:** `HTB{n3v3r_pl4ys_4lw4ys_w4tch3s}`

---

1. Challenge Overview

> *The Drowned Bell doesn't ask questions — which is exactly why Ferro Quicktongue runs his nightly game there. Three dented tankards, one bent copper chit, and a room full of off-duty gate clerks who really ought to know better than to bet against a man whose hands move faster than his mouth. [...] What catches Rin's eye is the table by the door, where a courier is running the exact same swap-and-shuffle — except the thing sliding under those tankards isn't a copper chit, it's a folded scrap bound for someone at Suncourt [...] working out where every marked tankard actually ends up, theirs included, is the only way to know which one held the real message.*

Behind the tavern flavour text this is a classic **permutation tracking** problem: apply a sequence of position swaps to a row of items, then report the final position of specific items.

2. Recon

The listed endpoint is a raw `host:port`, so a plain TCP probe was the first attempt — it returned nothing. A quick HTTP request instead revealed a Flask/Werkzeug app:

```
$ nc -zv -w 5 154.57.164.72 30699
Connection to 154.57.164.72 port 30699 [tcp/*] succeeded!

$ curl -s -i --max-time 15 http://154.57.164.72:30699/ | head -6
HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.14.0
Date: Sat, 25 Jul 2026 10:44:45 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 28469
Connection: close
```

The page title is simply `Coding` — this is HTB's standard **Coding** challenge harness: a Monaco editor, a language selector (`python`, `c`, `cpp`, `rust`), and a *Run Code* button. Stripping the markup out of the returned HTML gives the full problem statement.

Grepping the inline JavaScript for endpoints shows exactly one API route:

```
$ grep -oE "'/[a-zA-Z_/]+'|\"/[a-zA-Z_/]+\"" tankards.html | sort -u
"/run"
```

and the submission logic:

```
fetch("/run", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ code: code, language: document.getElementById("language-
select").value }),
})
  .then((response) => response.json())
  .then((data) => {
    if (data.challengeCompleted) {
      document.getElementById('flag').innerHTML = data.flag
      showSuccessModal();
    }
    ...
```

So the server compiles/runs the submitted source against a set of hidden test cases and returns the flag in JSON once every case passes. No browser needed — we can POST directly.

3. Problem Statement (extracted)

Input

- Line 1: three space-separated integers `N M Q`
- `N` — number of tankards, numbered `1..N`. Tankard `i` starts holding item `i`.
- `M` — number of swaps performed, in order.
- `Q` — number of starting positions to track.
- Next `M` lines: `a b` — the two *positions* whose contents are exchanged.
- Next `Q` lines: a single integer `p` — a starting position whose final resting place must be reported.

Output

Print `Q` lines. For each queried starting position `p`, print the position of the item that started at `p` after all `M` swaps have been applied in order.

Example

```
Input:
5 4 2
1 3
2 4
3 5
4 1
3
5
```

```
Expected output:
4
3
```

Walkthrough from the statement: the item starting at position 3 is moved to 1 by swap `(1,3)`; `(2,4)` and `(3,5)` don't touch position 1; then `(4,1)` moves it to 4. The item starting at 5 is untouched until `(3,5)` puts it at 3, and `(4,1)` doesn't touch position 3 — so it stays at 3.

4. The Trap

The naive reading is "just simulate the swaps on an array and then search for the item". That's where the challenge hides its cost:

- Swapping positions is `O(1)` per swap on an array `arr[]` where `arr[p]` = item currently at position `p`.
- But answering "*where did item `p` end up?*" from `arr[]` alone requires a linear scan — `O(N)` per query, i.e. `O(N·Q)` overall. With large hidden test cases that times out.

The other tempting-but-wrong approach is to track only the queried items and replay all `M` swaps per query (`O(M·Q)`) — same problem.

Key insight: maintain *both* directions of the mapping simultaneously.

- `arr[p]` — which item sits at position `p` (forward map)
- `pos[i]` — which position item `i` currently occupies (inverse map)

A swap of positions `a` and `b` updates both in constant time:

```
x = arr[a]; y = arr[b]
arr[a] = y;  arr[b] = x      # positions now hold the other item
pos[x] = b;  pos[y] = a      # and the items know their new home
```

Because item `i` starts at position `i`, the answer for a query `p` is simply `pos[p]` — an `O(1)` lookup. Total complexity: **`O(N + M + Q)`** time, `O(N)` memory. The two arrays are always exact inverses of one another, which is the invariant that makes the lookup free.

Note the small robustness detail: guard against `a == b`, which would otherwise still work with the code above but is worth short-circuiting.

5. Solution

Python, reading the entire stream with a single buffered read and splitting on whitespace (line-by-line `input()` is far too slow for large inputs):

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
```

```
    if not data:
        return
    idx = 0
    n = int(data[idx]); idx += 1
    m = int(data[idx]); idx += 1
    q = int(data[idx]); idx += 1

    # arr[p] = item currently at position p ; pos[i] = current position of item i
    arr = list(range(n + 1))
    pos = list(range(n + 1))

    for _ in range(m):
        a = int(data[idx]); idx += 1
        b = int(data[idx]); idx += 1
        if a == b:
            continue
        x = arr[a]; y = arr[b]
        arr[a] = y; arr[b] = x
        pos[x] = b; pos[y] = a

    out = []
    for _ in range(q):
        p = int(data[idx]); idx += 1
        out.append(pos[p])
    sys.stdout.write('\n'.join(map(str, out)))

main()
```

Local verification against the sample:

```
$ printf '5 4 2\n1 3\n2 4\n3 5\n4 1\n3\n5\n' | python3 sol.py
4
3
```

6. Submission

POST the source straight to `/run`:

```
import json, urllib.request

code = open('sol.py').read()
req = urllib.request.Request(
    "http://154.57.164.72:30699/run",
    data=json.dumps({"code": code, "language": "python"}).encode(),
    headers={"Content-Type": "application/json"})
print(urllib.request.urlopen(req, timeout=120).read().decode())
```

Response:

```
{"challengeCompleted":true,"flag":"HTB{n3v3r_pl4ys_4lw4ys_w4tch3s}","result":{}}
```

7. Flag

```
HTB{n3v3r_pl4ys_4lw4ys_w4tch3s}
```

8. Takeaways

- A raw `host:port` in a challenge description doesn't guarantee a raw TCP service — always try HTTP before assuming the service is silent.
- The HTB Coding harness is fully scriptable via a single `POST /run` with `{code, language}`; no browser automation required, which makes iterating on a solution much faster.
- The core algorithmic idea generalises well: whenever you need both "what's at slot X" and "where is element Y" under mutation, keep a permutation and its inverse side by side and update both on every operation. It turns an `O(N·Q)` scan into `O(1)` lookups.
- Fittingly for the story: Rin doesn't re-run the shuffle for each marked tankard — she keeps track of *every* cup at once, and the answer for any patron's bet is already in hand. Hence the flag: *never plays, always watches.*