CyberApocalypse 2026 — Crypto — Ancient Artifacts

Category: Crypto **Target:** `154.57.164.80:31433` (`server.py`) **Flag:** `HTB{zl1b_func710n5_r34lly_5houldn7_b3_us3d_1n_cryp70!}`

---

1. Challenge overview

The service uses two non-cryptographic checksums from `zlib` — **Adler-32** and **CRC-32** — as if they were secure hashes. The whole "*every archive leaves traces of its own construction*" flavour is a hint that these checksums leak their inputs and are trivially forgeable.

```
import zlib, functools, secrets, signal
signal.alarm(30)

def apply_rune(rune, *nums):
    return rune("".join(str(num) for num in nums).encode())

num = secrets.randbits(128)
h   = apply_rune(zlib.adler32, num)                 # h = adler32(str(num))

my_salt   = secrets.randbits(128)
your_salt  = int(input("your salt: "))
assert your_salt.bit_length() >= 128
salted = apply_rune(zlib.adler32, my_salt, num, your_salt)   # adler32(str(my_salt)-
print(f"{my_salt = }")
print(f"{salted = }")

nums = []
while True:
    option = int(input(MENU))
    if option == 2: break
    elif option == 1:
        your_num = int(input("n: "))
        assert your_num not in nums
        assert apply_rune(zlib.adler32, your_num) == h          # (A) every number h
        nums.append(your_num)
```

```
    else: exit("bye")

assert apply_rune(zlib.crc32, *nums) == h                        # (B) crc32 of the
assert functools.reduce(lambda a,b: a^b,
        [apply_rune(zlib.crc32, num) for num in nums]) == h      # (C) XOR of per-nu
print(open("flag.txt").read())
```

To win we must:

1. **Recover `h`** — but `h = adler32(str(num))` and `num` is a secret 128-bit integer. We only learn `my_salt` and `salted = adler32(str(my_salt)+str(num)+str(your_salt))`.
2. Provide distinct integers `nums` such that:
3. **(A)** `adler32(str(n)) == h` for every `n`,
4. **(B)** `crc32(str(n1)+str(n2)+…) == h`,
5. **(C)** `crc32(str(n1)) ^ crc32(str(n2)) ^ … == h`.

Note `num` is only ever used to compute `h`; we never need `num` itself, only the 32-bit value `h`.

---

2. Step 1 — Recovering `h` via `adler32_combine` inversion

Adler-32 has a well-known composition rule (`adler32_combine` in zlib): the checksum of `X‖Y` can be computed from the checksums of `X` and `Y` plus `len(Y)`. Writing `adler(s) = a + (b<<16)` with `a,b < BASE = 65521`:

```
combine(adler(X), adler(Y), len(Y)):
    Ac = (aX + aY − 1)                 (mod BASE)
    Bc = (bX + bY + (lenY mod BASE)·(aX − 1))   (mod BASE)
```

This map is **linear and invertible**. Given the combined value and one operand (with the length of the other), we can solve for the missing operand:

```
def solve_a1(combined, a2, l2):   # recover first operand X from combined & Y
    rem = l2 % BASE
    Ac, Bc = combined & 0xffff, (combined >> 16) & 0xffff
    a2a, a2b = a2 & 0xffff, (a2 >> 16) & 0xffff
    a1a = (Ac − a2a + 1) % BASE
    a1b = (Bc − a2b − rem*(a1a − 1)) % BASE
    return a1a | (a1b << 16)
```

```
def solve_a2(combined, a1, l2):   # recover second operand Y from combined & X
    rem = l2 % BASE
    Ac, Bc = combined & 0xffff, (combined >> 16) & 0xffff
    a1a, a1b = a1 & 0xffff, (a1 >> 16) & 0xffff
    a2a = (Ac - a1a + 1) % BASE
    a2b = (Bc - a1b - rem*(a1a - 1)) % BASE
    return a2a | (a2b << 16)
```

`salted = combine( adler(P1‖P2), adler(P3), len(P3) )` where `P1=str(my_salt)`, `P2=str(num)`, `P3=str(your_salt)`. Since we know `P1`, `P3` and `salted`:

1. `X = adler(P1‖P2) = solve_a1(salted, adler(P3), len(P3))`.
2. `X = combine( adler(P1), h, len(P2) )` ⟹ `h = solve_a2(X, adler(P1), len(P2))`.

The only unknown is **`len(P2) = len(str(num))`**. Crucially, the low-16 (the `a`-part) of `h` is **independent of that length** — only the high-16 (`b`-part) depends on it. So we brute-force the length `L` (a 128-bit integer has ≈37–39 decimal digits) and get one `h` candidate per `L`.

Disambiguating the length

The recovered `a`-part fixes the total **byte sum** of `str(num)`: `Sbytes = (a − 1) mod BASE`. For an `L`-digit decimal number (leading digit ≥ 1, ASCII bytes 48–57) this forces `48·L < Sbytes ≤ 57·L`, and the achievable range of the Adler `b`-value is a small interval `[b_min(L), b_max(L)]`. Filtering candidate `L` values against this interval leaves **exactly one** `h` in practice (verified over 200 random simulations — always a unique, correct candidate).

> **Key insight:** the salt "protection" is useless — Adler-32 concatenation is invertible, so `h` (and the *byte-sum* of the secret) leak directly from the salted checksum.

---

3. Step 2 — Forging a single number that satisfies all three checks

We don't need many numbers. If we submit a **single** number `s`, then: - (A) needs `adler32(str(s)) == h`, - (B) `crc32(str(s)) == h` (concatenation of one element), - (C) `crc32(str(s)) == h` (XOR of one element).

So **(B)** and **(C)** collapse to the same condition. We just need **one decimal string `s` with `adler32(s) == h` AND `crc32(s) == h`**. The elegant part is doing both simultaneously.

Layout

```
s = "1"  +  [ A-region: many digits 0–9 ]  +  [ C-region: 64 blocks of 4 digits ]
     └leading (no leading zero)                 └used to fix CRC-32
```

Fixing Adler-32 with the A-region

Adler-32 over bytes `B_j` (here `B_j = 48 + digit`) is:

```
a = (1 + Σ B_j)              mod BASE
b = (L + Σ (L−j)·B_j)        mod BASE
```

Two linear congruences mod `BASE`. With a few thousand free digits we can hit any `(a,b)`: - Choose the **total length** so the required digit-sum residue is reachable in a compact region (keeps the number well under Python's 4300-digit `int()` limit — the server calls `int(input())`, so an oversized number would just crash it). - Solve the **digit-sum** equation, then place the digits to hit the required **weighted-sum (moment)** residue via a greedy high-to-low feasibility construction.

Fixing CRC-32 with adler-neutral toggles

The trick: each **4-digit block toggles between `"0110"` and `"1001"`**. Both patterns have the **same digit sum (2)** and the **same positional weighted sum**, so toggling a block is **Adler-neutral** — it never disturbs the Adler value we just fixed.

CRC-32 is affine over GF(2), so flipping block *i* changes the CRC by a fixed vector `v_i = crc(baseline_with_block_i_toggled) ⊕ crc(baseline)`, and multiple toggles compose by XOR. We solve the GF(2) system

```
⊕_{i∈T} v_i = h ⊕ crc(baseline)
```

with Gaussian elimination (64 blocks ⇒ comfortably rank-32), then set the chosen blocks to `"1001"`. Result: `crc32(s) == h` while `adler32(s)` stays `== h`.

The produced number is ≈3.5k digits — valid and fast to build (< 0.1 s).

---

4. Putting it together

Protocol interaction: 1. Send any `your_salt` with `bit_length() ≥ 128`. 2. Parse `my_salt` and `salted`. 3. Recover the unique `h`. 4. Build the single forged number `s`. 5. Menu: `1` → send `s` → `2` (stop) → read the flag.

One important implementation detail: on the remote, the `salted` line and the menu arrive in the **same TCP burst**, so the socket reader must keep a persistent buffer across reads (a naive per-call `recv_until` deadlocks).

Running `exploit.py` against the live service returned the flag on the first attempt:

```
[*] byte-sum S=2058, 1 candidate h(s): [2686519307]
[*] using h=2686519307, built number with 3527 digits
[*] server response: '\nHTB{zl1b_func710n5_r34lly_5houldn7_b3_us3d_1n_cryp70!}\n'
```

---

5. Flag

```
HTB{zl1b_func710n5_r34lly_5houldn7_b3_us3d_1n_cryp70!}
```

6. Takeaways

- **Adler-32 is not a hash.** Its concatenation rule (`adler32_combine`) is linear and invertible, so salting `adler32(salt‖secret‖salt2)` leaks `adler32(secret)` — and the byte-sum of the secret — outright.
- **CRC-32 is linear over GF(2)** and forgeable to any target with a handful of controllable bytes.
- Combining the two is easy once you find **checksum-neutral edits**: the `"0110" ↔ "1001"` block toggle preserves Adler-32 exactly while giving free GF(2) control over CRC-32, letting us satisfy both checks in one string.
- Non-cryptographic checksums must never be used where collision/second-preimage resistance is assumed — exactly what the flag says.