Cyber Apocalypse 2026 — Crypto — Ashbyte Arcade

Flag: `HTB{tw0_r0und5_0f_5n0w_4nd_1mp0551bl3_d1ff5}`

Target: `154.57.164.68:31018`

---

1. Challenge overview

The service is a retro-arcade cabinet. When your runner "crashes" it prints an encrypted *restore tape*, and you can *load* a tape back. If the tape decrypts to a **winning save state**, the vault opens and hands over the flag.

Provided files:

File	Role
`app.py`	Flask app: session, `/api/start`, `/api/death` (encrypt), `/api/load` (decrypt + win check)
`crypto.py`	`SnowCipher` — a custom block cipher, plus `compute_lrc`
`templates/`, `static/`	The game front-end (irrelevant to the crypto)

1.1 The cipher — `SnowCipher`

`SnowCipher` is **AES reduced to 2 rounds**, with one twist: *both* rounds run MixColumns (normally the final AES round omits it). Everything else — S-box `B`, ShiftRows, MixColumns, and the standard AES-128 key schedule (truncated to the first 3 round keys `K0, K1, K2`) — is textbook AES.

```
encrypt_block(P):
    s = ARK(P, K0)
    s = MC(SR(SB(s)));  s = ARK(s, K1)      # round 1
```

```
    s = MC(SR(SB(s)));  s = ARK(s, K2)      # round 2
    return s
```

So a single 16-byte block is:

```
C = ARK( MC(SR(SB( ARK( MC(SR(SB( ARK(P,K0) ))), K1) ))), K2)
```

The key is `secrets.token_bytes(16)` — a fresh random 128-bit key **per session**.

1.2 The oracles

- **`/api/death`** — an **encryption oracle**. You submit a 16-byte save state (`state_hex`); it is validated, and if accepted it is encrypted and the ciphertext returned. Encryption uses ECB with PKCS#7, so a valid save becomes **32 bytes**: `enc(state) || enc(0x10*16)` (the padding block is constant per key).
- **`/api/load`** — a decryption oracle that reports whether the tape decrypts to a winning state, returning the flag if so.

The catch: each session starts with **16 lives**, and *every* `/api/death` (even a rejected probe) burns one life. So you get **at most ~16 chosen-plaintext pairs per key**, and a new session resets to a brand-new random key. Classic Square/integral attacks (256 texts) are therefore off the table.

1.3 Plaintext constraints

`require_state_format` forces the save to be well-formed:

- `st[0] = SIG = 0x4E` (constant), HP∈1..9, sector∈1..9, x∈1..18, y∈1..13, deaths≤31, `st[8]^st[9]^st[10]^st[11] = 0xA7`;
- `st[12]` = track seal, `st[14]` = flag seal, `st[15]` = LRC (all derived).

The winning state also refuses to be encrypted (`is_winning_state` is blocked in `/api/death`), so we can never simply ask the oracle to encrypt the goal. We must recover the key and forge `enc(winning_state)` ourselves.

The winning state is fully determined:

```
4e 05 09 12 0d 80 96 98 c3 7a b0 ae 52 00 7f 59
```

Crucially it satisfies `SIG = 0x4E` just like every other valid save — remember this, it matters.

---

2. Cryptanalysis — differential key recovery on 2-round AES

Two AES rounds fully diffuse (flipping one plaintext byte changes all 16 ciphertext bytes), so we can't isolate key bytes from a single pair. But 2 rounds is still far too few, and the structure collapses under a **differential meet-in-the-middle**.

Let us name the internal wires for a plaintext `P`:

```
x = SB(P ⊕ K0)                    # after round-1 S-box (byte-wise)
y = MC(SR(x)) ⊕ K1                # round-1 output
z = SB(y)                         # after round-2 S-box (byte-wise)
C = MC(SR(z)) ⊕ K2                # ciphertext
```

2.1 Step 1 — `Δz` for free (K2 cancels)

Take two plaintexts `P1, P2` with ciphertexts `C1, C2`. Because `MC`, `SR`, and `⊕K2` are all *linear*, the last round key cancels in the difference:

```
ΔC = C1 ⊕ C2 = MC(SR(Δz))     ⇒     Δz = SR⁻¹( MC⁻¹( ΔC ) )
```

We recover the difference right after the **second S-box** with **no key knowledge at all**. This is the crack that makes everything else cheap.

2.2 Step 2 — 4 independent super-boxes → recover `K0`

ShiftRows in round 1 groups the state into four independent 32-bit *super-boxes*, each fed by one diagonal of the plaintext:

super-box	plaintext positions (= K0 positions)
0	0, 5, 10, 15
1	3, 4, 9, 14
2	2, 7, 8, 13
3	1, 6, 11, 12

For super-box `g`, guess its 4 bytes of `K0` (2^{32}). For a pair we then know the round-1 S-box output difference `Δx` on those positions, and since `MC∘SR` is a fixed known linear map `L_g`,

```
Δy_group = L_g · Δx        (4 bytes)
```

The bytes `(Δy_group[k], Δz_group[k])` must be a **valid S-box differential**, i.e. the Difference Distribution Table entry `DDT[Δy][Δz] > 0`. A wrong `K0` guess fails this with probability ≈ 15/16 per pair, so **~7 pairs uniquely pin each super-box's `K0` diagonal**. Four super-boxes × 2^{32} is ~2^{34} cheap operations — a couple of minutes in C.

The SIG byte. Position 0 is always `0x4E`, so its input difference is always 0 → `K0[0]` never affects any differential and is **unrecoverable**. That's fine: `x[0] = SB(0x4E ⊕ K0[0])` is a *constant* for every valid save, so its effect is a fixed additive constant that is simply **absorbed into `K1`**. Since the winning state is *also* `SIG = 0x4E`, the forged encryption is still exact. We just fix `K0[0] = 0`.

2.3 Step 3 — `K1` byte-wise, then `K2` directly

With `K0` known, `Y0(P) = MC(SR(SB(P ⊕ K0)))` is computable, and `y = Y0(P) ⊕ K1`. Since `z = SB(y)` is byte-wise and we know `Δz`, each key byte solves a per-byte differential:

```
for each byte i:   SB(Y0(P1)[i] ⊕ k) ⊕ SB(Y0(P2)[i] ⊕ k) = Δz[i]
```

A handful of pairs intersect to a unique `K1`. Finally `K2` is a direct subtraction from any known pair:

```
z  = SB(Y0(P) ⊕ K1)
K2 = C ⊕ MC(SR(z))
```

We now hold `K0, K1, K2` — enough to encrypt *any* `SIG=0x4E` block, no master key needed.

2.4 Step 4 — forge and win

```
forged      = enc_with_rk(winning_state, K0, K1, K2)   # our own re-implementati
padding_blk = second 16 bytes of any death tape         # constant enc(0x10*16)
tape        = forged || padding_blk                     # 32 bytes
POST /api/load { ciphertext: tape } -> flag
```

---

3. Exploit

The full solver is `solver.py`. It:

1. opens a session and sends 14 distinct valid saves to `/api/death`, collecting 14 `(plaintext, ciphertext-block)` pairs and the constant padding block;
2. emits a small C program (S-box differential brute force) to recover `K0` per super-box in ~2 min, then finishes `K1` / `K2` in Python;
3. verifies the recovered round keys reproduce all 14 observed encryptions;
4. forges `enc(winning_state)`, appends the padding block, and loads the tape.

Run:

```
python3 -m venv .venv && .venv/bin/pip install pycryptodome flask
.venv/bin/python solver.py --host 154.57.164.68 --port 31018 -n 14
# (self-test against a random local key: .venv/bin/python solver.py --selftest)
```

Output (abridged):

```
[*] collected 14/14 (lives left 2)
[*] recovering round keys (C brute ~2 min)...
[*] superbox 0: 1 K0-diagonal survivor(s)
[*] superbox 1: 1 K0-diagonal survivor(s)
[*] superbox 2: 1 K0-diagonal survivor(s)
[*] superbox 3: 1 K0-diagonal survivor(s)
[+] K0=004404de78e12e9b99ace689cf30feaa
[+] K1=99baed9b6b1e86cff2b260463d829eec
[+] K2=02f4667369eae0bc9b5880faa6da1e16
[*] forged winning ciphertext: 81c5883dd57b1d9a8c8c2f8c082e53e326974751ebb5a6fe5
...
[+] FLAG: HTB{tw0_r0und5_0f_5n0w_4nd_1mp0551bl3_d1ff5}
```

(K0[0] shows as `00` because it is the unrecoverable/absorbed SIG byte, exactly as expected.)

---

4. Flag

```
HTB{tw0_r0und5_0f_5n0w_4nd_1mp0551bl3_d1ff5}
```

5. Key takeaways

- **Round count is security margin.** AES is safe at 10 rounds; at 2 it dies to a single differential meet-in-the-middle needing only a handful of pairs.
- **The last round key is free in a difference** — subtract two ciphertexts and every linear layer (incl. the final key add) cancels, peeling a whole round.
- **ShiftRows creates independent super-boxes**, turning a 2^{128} key search into four independent 2^{32} searches.
- **You don't always need the true key.** A constant plaintext byte (`SIG`) makes `K0[0]` invisible, but it is absorbed into `K1` ; since the goal shares that constant, the forgery is still exact.