Old Cassette

THEM?!CTF 2026 - Complete Reverse Engineering Writeup

Challenge file: main.bin (3,282-byte CHIP-8 ROM)

Verified flag: THEM?!CTF{0LD_T4P3_N3V3R_D1E5K7}

This writeup walks through the challenge from the ground up. We identify the binary, map the program layout, explain the tiny PRNG and delay logic, and then show how the huge-looking wait collapses into a small and fully tractable cycle problem.

1. Challenge Overview

Old Cassette is a CHIP-8 reverse-engineering challenge dressed up like an old tape-era ROM, and that theme is doing real work. CHIP-8 historically lived on small hobbyist systems that loaded programs from simple media, so the cassette flavor is a neat way to hide a compact but carefully layered routine that reveals the flag one character at a time.

At a high level, the ROM does three things:

1) it seeds a tiny two-register PRNG,
2) it advances that PRNG through nested delay loops,
3) it uses the resulting state to decrypt and display the next glyph.

The key insight is that the PRNG state space is only 16 bits. That means repetition is guaranteed. Once you recognize the cycle, the intimidating delay becomes a modular arithmetic problem instead of a brute-force simulation.

Item	Value
Category	Reverse Engineering
Platform	CHIP-8 virtual machine
Input	main.bin
Core obstacle	Artificially huge PRNG delay
Key insight	State cycle is only 16 bits
Outcome	32-character flag reveal

2. First Triage: Identifying the Binary

A file-type check is almost useless here because the ROM has no extension. The fastest way to identify it is to look at the raw bytes. CHIP-8 opcodes are two bytes wide, and the first instruction in a normal program is often CLS (00E0), which clears the display.

```
$ xxd main.bin | head -1
00000000: 00e0 1280 a6a6 a6a6 a6a6 a6a6 a6a6 a6a6  ................
```

That first word, 00E0, is CHIP-8 CLS. The next word, 1280, is a jump to 0x280. Everything between 0x204 and 0x27E is not code; it is data bytes that the interpreter skips over. That layout is classic CHIP-8 organization: a tiny entry stub, then a jump over inline data, then the real program body.

The repeated A6 bytes become useful later. In the HTML writeup, they are treated as sprite-like data, and the UI demo even uses 0xA6 as a dummy plaintext byte to illustrate the XOR relationship. That is a strong hint that the challenge is not about heavy encryption, but about a very small and regular data path.

3. CHIP-8 Primer for the Challenge

CHIP-8 is a tiny interpreted virtual machine. It has 4K of memory, 16 8-bit registers V0 through VF, a 64x32 monochrome display, a stack, a delay timer, and a small instruction set of roughly 35 opcodes. Programs load at address 0x200. Because the machine is so constrained, reverse engineering a CHIP-8 ROM usually means reading the control flow directly from opcodes instead of relying on broad compiler patterns.

In this challenge, the author uses those CHIP-8 constraints very effectively. The ROM does not need to be complicated to be annoying. A few nested loops, a few register transforms, and a large wait counter are enough to make the correct solution look much harder than it really is.

Two CHIP-8 details matter most here:

The VF register is used as the carry/borrow flag in arithmetic and shifts. The ROM leans on that behavior inside the PRNG mixer.

The draw instruction DRW reads sprite bytes from memory at I and XORs them onto the display. If the sprite bytes are encrypted before being written to I, the display output becomes the decryption target.

4. Memory Layout and Entry Path

The entry path is deliberately sparse. At 0x200 the program clears the screen and jumps to 0x280, which cleanly separates control code from data.

```
0x200: CLS
0x202: JP 0x280
0x204-0x27E: inline data bytes
0x280: delay gate / wait logic
0x2C0: PRNG subroutine
0x300: state diffusion and storage
0x322: sprite-bank selector
```

That is the first structural clue: this is not a sprawling application. It is a small set of routines wired together around a data table and a timed reveal loop. Once you can label those pieces, the rest of the ROM becomes much easier to read.

5. The Delay Gate: Why the Program Feels Slow

The code at 0x280 acts as the outer gate for the delay mechanism. It checks several loop counters in sequence, and if any of them are still non-zero it calls the PRNG update routine again before decrementing the counters.

```
0x280: JP 0x900          ; jump into main display routine
0x282: if V9 != 0: continue
```

```
0x286: if VC != 0: continue
0x28A: if VD != 0: continue
0x28E: if VE != 0: continue
0x292: RET

0x294: CALL 0x2C0        ; advance PRNG once
0x296: V9 -= 1
0x29C: VC -= 1
0x2A2: VD -= 1
0x2A8: VE -= 1
... and the loop repeats
```

The counters are initialized in different blocks to large values. In the repository writeup, the combined effect is described as eventually reaching roughly 1.1 trillion PRNG steps before the final reveal completes.

The lesson is simple and useful: when a CTF binary seems to spend forever inside nested loops, do not brute-force the loop unless the state machine truly demands it. Look for periodicity, finite state, and opportunities to reduce time with math.

6. Dissecting the PRNG

The heart of the challenge is the subroutine at 0x2C0. The helper script solve.py in the repository reproduces it exactly. The routine works on two 8-bit registers, VA and VB, and mixes them with a small lookup table plus a bit-rotation stage.

```
# conceptually:
v0 = mem[0x800 + VB] ^ VB ^ CONSTS[VB >> 6]
carry = 1 if VB + v0 > 255 else 0
VB = (VB + v0) & 0xFF
VA = (VA + carry) & 0xFF

# then copy the old state into (v2, v3)
# rotate that pair left by 5 bits total
# and XOR the result back into the live state
VA ^= rotated_v2
VB ^= rotated_v3
```

The solver shows the exact logic, including the 4-entry constant table [A9, 5C, D3, 76]. The important part is that the PRNG is deterministic, 16-bit wide, and fully recoverable once the CHIP-8 instructions are translated back into state transitions.

The rotation stage is especially important because it is not random noise. It is a diffusion step. By rotating and XORing the pre-update state back into the registers, the routine spreads entropy across both bytes while still staying completely deterministic.

From a reverse-engineering perspective, the important takeaway is this: the PRNG looks custom and messy, but it is actually quite small. Once you translate the CHIP-8 opcodes, it becomes a normal state machine that you can model exactly.

7. The Cycle Problem: Finite State Always Wins

The solver script starts from the seed read directly from the ROM entry path: (A7, C3). It then walks the state transition function until a state repeats. Because there are only 65,536 possible (VA, VB) pairs, repetition is guaranteed by the pigeonhole principle.

```
seed = (A7, C3)
first repeat at step = 329
cycle length = 34
repeat state = (C7, 69)
```

That result is the entire trick. The ROM does not require you to simulate a trillion updates. It only requires you to notice that the PRNG orbit is eventually periodic, then reduce the huge delay to a position inside that cycle.

```
1,095,216,660,225 mod 34 = 17
```

So the enormous delay collapses to 17 effective steps inside the repeating portion of the orbit. At that point, the ROM is no longer hiding state in time. It is just asking you to do a small amount of modular bookkeeping.

This is exactly the kind of reduction CTF reverse engineering rewards: the implementation is noisy, but the state space is tiny.

8. Understanding the Sprite Selection and Decryption

After the PRNG advance, the ROM prepares a character for display. The relevant code masks VA with 0x07, uses that to select a sprite bank, offsets I by a per-character constant, and stores a byte transformed using both VA and VB.

```
V0 = VA & 0x07
CALL 0x322           ; choose a sprite bank based on V0
I += offset
store selected byte
V9 = stored byte XOR VA XOR VB
```

The precise bank offsets differ from block to block, but the pattern stays consistent. The ROM repeatedly resets V9/VC/VD/VE, waits through the delay gate, selects a sprite bank, applies the PRNG-derived mask, and reveals the next byte.

That is why the final artifact is a 32-character flag rather than a graphic puzzle or a long decryption transcript. Each cycle of the main routine reveals exactly one character. Once the cycle is understood, the output becomes predictable.

9. Verification With the Provided Solver

The repository includes solve.py, which is a major quality-of-life clue. It loads the ROM into a 4K CHIP-8 memory image, reimplements the PRNG transition, and walks the orbit until repetition. Running it reproduces the exact cycle data used in the writeup.

```
$ python3 solve.py main.bin
seed = (A7, C3)
first repeat at step = 329
cycle length = 34
repeat state = (C7, 69)
```

That script is the reproducibility anchor for the whole solution. It shows that the challenge author expected the PRNG to be modeled mathematically, not emulated at full speed. It also makes the final flag verification completely deterministic.

A second verification clue appears in the bundled HTML writeup itself, where the final reveal logic decodes a base64 string into the same flag. That is not the primary solution path, but it is a very nice consistency check.

10. Final Flag

After reducing the delay and walking the PRNG states through the display loop, the recovered plaintext is the 32-character flag below.

```
THEM?!CTF{0LD_T4P3_N3V3R_D1E5K7}
```

The length is exactly 32 characters, matching the ROM's 32-stage reveal structure. That symmetry is a good final sanity check: the challenge output and the program structure line up cleanly.

11. Key Takeaways

1) Do not trust apparent time complexity in a CTF binary. Nested loops can hide a finite cycle that collapses under modular arithmetic.

2) For CHIP-8, the first two bytes often reveal the shape of the whole program. Here, 00E0 and 1280 immediately identify the format and the real entry point.

3) Always separate code from data when a ROM jumps over a block of bytes. The challenge relies on that distinction.

4) When a solver script is shipped with the challenge, treat it as an oracle for the state transition, not as a crutch. It usually documents exactly what the author wants you to understand.