# Cyber Apocalypse 2026 — Reversing — Secondhand

**Flag:** `HTB{WHY_S00_345Y_H0N}`

**Category:** Reversing · **File:** `./secondhand` (ELF 64-bit x86-64, dynamically linked, not stripped)

---

## TL;DR

The binary presents two elaborate, fully-solvable crypto checks — a 28-byte rolling-XOR routine (`b_roll`) and a 10-round AES-lookalike block cipher (`threat_level_midnight`). Both are decoys: solving either just prints "*That is the line we cut into the trailer … It is not in the film.*" One of them even hands you a plausible-looking fake flag, `HTB{secondhand_rough_cut_v3}`.

The real check only fires for an input of **exactly 8 bytes**. It decrypts a 38-byte stub into freshly-`mmap`'d RWX memory and calls it, then compares the stub's output against a "hash" computed by a helper that **jumps into the middle of an instruction**. At that misaligned offset the code is not a hash loop at all — it is `cmp rax, 0xb8491337c0debabe ; jne +1 ; ret`, which short-circuits the whole comparison. So the win condition collapses to inverting one invertible 64-bit function.

That 8-byte key then doubles as a `splitmix64` seed for the real flag's keystream.

---

## 1. Recon

Two exported symbols and a suspicious import list:

```
$ file secondhand
secondhand: ELF 64-bit LSB executable, x86-64, ... dynamically linked, not stripped
$ nm secondhand
0000000000401370 T b_roll
00000000004014f0 T threat_level_midnight
```

`mmap` + `mprotect` + `munmap` alongside `ptrace` and `/proc/self/status` is the whole challenge in one line: there is self-modifying code, and it is guarded by anti-debug.

`strings` gives us the narrative — and the hint list, which is worth reading twice because every one of them points at a decoy:

```
editing log: rolling key looks random enough on the scopes, shipping it
do not retune the round multiplier, the whole reel was graded against it
the rotor pass is what makes the static read as noise, leave it in
note to self: the loop is the only part of this cut that was hand cut
```

"Rolling key" is `b_roll`. "Round multiplier" and "rotor pass" are `threat_level_midnight`. "**The loop is the only part of this cut that was hand cut**" is the one that matters — and the prose hint spells out the technique outright:

> *it only feeds clean if you start the page from the inner crease, not from the top. Feeding it from the top is why it keeps jamming.*

That is a description of a misaligned instruction stream. Disassemble from the middle of the instruction, not from its first byte.

## 2. Control flow of `main` (0x401100)

```
int len = read_line(stdin, buf, 0x100);   // 0x4013e0, returns length sans '\n'
if (len < 0) { fputs("nope\n", stderr); return 1; }

if (len == 8 && check8(buf, 8))           // 0x401a40  ⟵ the real path
    return 0;                             //   (prints the flag itself)

fputs("This is a rough cut. ...");        // 0x402060
fputs("Facilities, second floor printer. ...");

if (debugger_present())                   // 0x401330
    fputs("There is a camera in the room. ...");
else if (b_roll(buf) || threat_level_midnight(buf, len))
    fputs("That is the line we cut into the trailer. ... It is not in the film.");
return 0;
```

Note the shape of this: the *only* branch that does not lead back to a narrative string is `check8`, and it is gated behind `len == 8`. Both named, symbol-exported functions lead to the same "not in the film" dead end.

The anti-debug at `0x401330` is standard and only affects the decoy branch: it greps `/proc/self/status` for a non-zero `TracerPid:`, then falls back to `ptrace(PTRACE_TRACEME, 0,`

`0, 0) == -1`. It never guards the real path, so it is not even an obstacle — just another misdirection.

## 3. The decoys (and why you should walk away from them)

### 3.1 `b_roll` — 28 bytes, rolling key

```
mov ecx, 0x5a                       ; k
.loop:
  movzx r8d, byte [rbx+rdx]         ; c = buf[i]
  lea   eax, [r8+rsi]               ; c + 31*i
  rol   al, 3
  xor   eax, ecx                    ; ^= k
  cmp   byte [rdx+0x402040], al     ; must match target[i]
  add   ecx, r8d                    ; k += c        ⟵ key rolls with the plaintext
  add   esi, 0x1f
```

`target[i] = rol3((c + 31*i) & 0xff) ^ k`, with `k` starting at `0x5a` and accumulating the plaintext. Because `k` only ever depends on *already recovered* bytes, this peels off left to right with no search:

```
k = 0x5a
for i in range(28):
    c = (ror3(target[i] ^ (k & 0xff)) - 31*i) & 0xff
    out.append(c); k += c
```

Result: `HTB{secondhand_rough_cut_v3}` — a perfectly formatted flag that is **wrong**. Feed it back to the binary and it tells you so: "*It tested well. It is not in the film.*"

### 3.2 `threat_level_midnight` — 16 bytes, a hand-rolled AES

This one is a genuine time sink, and mostly SSE-vectorised so it reads badly. It:

1. Builds a 256-byte S-box with `packed byte` arithmetic — the shift/sub chain `2x → 4x → 8x → 7x → 14x → 28x → 27x` then `pxor 0x63` computes `SBOX[i] = (27*i) ^ 0x63`.
2. Generates 176 round-key bytes (11 × 16) by iterating an FNV/splitmix hybrid from `0x9E3779B97F4A7C15` — the "round multiplier" the editing log tells you not to retune.
3. Runs 10 rounds of `AddRoundKey → SubBytes → ShiftRows → rotor`, where `ShiftRows` is the byte permutation `[0,5,10,15,4,9,14,3,8,13,2,7,12,1,6,11]` (AES's, in column-major order) painstakingly open-coded as `shl` / `or` chains, and the "rotor pass" is `b ^= s; rol b, (i + round); s += b` with 32-bit carry in `s`.
4. Compares against 16 bytes at `0x402510`.

Every stage is invertible, so you can peel all 10 rounds and land on the unique preimage:

```
cd a5 0d ba 01 b2 cb 55 78 ce 21 79 88 32 f0 3b
```

Sixteen bytes of noise. It satisfies the check, and the check prints… the same "not in the film" line. All that work buys a decoy. This is the trap the challenge is built around: it is by far the most *impressive-looking* code in the binary, so it is where you naturally spend your time.

## 4. The real path: `check8` (0x401a40)

```
uint64_t x = 0;
for (i = 0; i < 8; i++) x |= (uint64_t)buf[i] << (8*i);   // little-endian

uint64_t expect = run_hidden_stub(x);        // 0x401c50
uint64_t got    = weird_hash(buf, 8, expect);// 0x401d90
if (expect != got) return 0;
/* ... prints the flag ... */
```

### 4.1 The RWX stub (0x401c50)

```
mov  rbp, [0x402540]                  ; n = 0x26 (38)
mmap(NULL, n, PROT_READ|WRITE, MAP_PRIVATE|ANON, -1, 0)
; decrypt 38 bytes read BACKWARDS from 0x402560
;   stub[i] = rol3(enc[n-1-i]) ^ key7[i % 7]      key7 @ 0x402535
memcpy(page, stub, n)
mprotect(page, n, PROT_READ|PROT_EXEC)
call page                             ; rdi = x
munmap(page, n)
```

Decrypting by hand:

```
enc  = rd(0x402560, 38)
key  = rd(0x402535, 7)
stub = bytes(rol8(enc[37-i], 3) ^ key[i % 7] for i in range(38))
```

```
4889f8                    mov     rax, rdi
49bb efbeaddea5a5a5a5     movabs  r11, 0xa5a5a5a5deadbeef
4c31d8                    xor     rax, r11
48c1c011                  rol     rax, 17
49bb b301000000010000     movabs  r11, 0x100000001b3
490fafc3                  imul    rax, r11
48f7d0                    not     rax
c3                        ret
```

So `expect = ~(rol64(x ^ 0xa5a5a5a5deadbeef, 17) * 0x100000001b3)` — a composition of XOR, rotate, odd multiply and NOT. **Every step is a bijection**, so it inverts exactly.

### 4.2 The misaligned jump (0x401d90) — the actual trick

At first glance `weird_hash` looks like it computes an FNV-style hash of the input, which would leave us solving `hash(buf) == stub(x)` — a 64-bit fixpoint over the same 8 bytes we control, with no clean algebraic handle. That is the intended dead end. Look at how it dispatches:

```
0x401d90:  mov  rax, rdx              ; rax = expect
           push rbp
           mov  r11d, esi             ; len = 8
           and  r11d, 0x1f
           xor  r11d, 0xe             ; 8 ^ 14 = 6
           lea  rdx, [rip+0x2a]       ; 0x401dd0
           add  rdx, r11              ; ---> 0x401dd6, NOT 0x401dd0
           push 0x401dc3              ; two staged return addresses
           push 0x401dbd
           jmp  rdx
```

The computed offset is `(len & 0x1f) ^ 0xe`, which for `len == 8` is **6**. objdump shows the function starting at `0x401dd0`:

```
0x401dd0:  55                          push rbp
0x401dd1:  48 89 e5                    mov  rbp, rsp
0x401dd4:  48 b8 49babebadec03713      movabs rax, 0x1337c0debabeba49   ; ⟵ hash IV
0x401dde:  49 b8 4c39d07501c331c0      movabs r8,  0xc031c30175d0394c   ; ⟵ multiplier
0x401de8:  31 c9                       xor  ecx, ecx
0x401dea:  ...                         ; hash loop
```

But we never enter at `0x401dd0`. Start the page from the inner crease — six bytes in, in the middle of that first `movabs` — and the immediates re-decode as opcodes:

```
0x401dd6:  49 ba bebadec0371349b8      movabs r10, 0xb8491337c0debabe
0x401de0:  4c 39 d0                    cmp    rax, r10
0x401de3:  75 01                       jne    0x401de6
0x401de5:  c3                         ret                 ; ⟵ taken when equal
0x401de6:  31 c0                       xor    eax, eax     ; else: hash from 0
0x401de8:  ...                         ; falls into the real hash loop
```

Those two staged `push`es make both exits work. `rax` still holds `expect` on entry, so:

- **`expect == 0xb8491337c0debabe`** → the `ret` at `0x401de5` unwinds via `0x401dbd` (`add rsp,8 ; pop rbp ; ret`) and returns `expect` **unchanged**. The caller's `expect ≠ got` comparison then trivially passes.

- otherwise → `rax` is zeroed, the hash loop runs, `pop rbp` swallows one staged address and `ret` lands on `0x401dc3`, returning a hash that will essentially never match.

So the entire check reduces to a single equation in an invertible function:

```
~(rol64(x ^ 0xa5a5a5a5deadbeef, 17) * 0x100000001b3) == 0xb8491337c0debabe
```

### 4.3 Solving it

```
M64  = (1 << 64) - 1
K, M = 0xa5a5a5a5deadbeef, 0x100000001b3
t = (~0xB8491337C0DEBABE) & M64
t = (t * pow(M, -1, 1 << 64)) & M64      # undo the odd multiply
x = ror64(t, 17) ^ K                     # undo rotate + xor
# x = 0x28382f8d1780680f  →  0f 68 80 17 8d 2f 38 28
```

The key is 8 **raw** bytes, not ASCII — `0f 68 80 17 8d 2f 38 28`. That is fine: `fgets` only cares about `\n`, and none of these bytes is `\n` or `\0`.

## 5. Recovering the flag

On success the binary prints `H4D5_UP_Y0U_M4D3_IT` (a taunt, not the flag) and then decrypts 21 bytes from `0x402520` with a keystream seeded by **the same** `x` — a textbook `splitmix64`, consumed 8 little-endian bytes at a time:

```
s, ks = 0x28382f8d1780680f, bytearray()
while len(ks) < 21:
    s = (s + 0x9E3779B97F4A7C15) & M64
    z = ((s ^ (s >> 30)) * 0xBF58476D1CE4E5B9) & M64
    z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) & M64
    ks += ((z ^ (z >> 31)) & M64).to_bytes(8, 'little')

flag = bytes(a ^ b for a, b in zip(rd(0x402520, 21), ks))
```

```
HTB{WHY_S00_345Y_H0N}
```

## 6. Verification

The solver derives the key purely algebraically, so it is worth confirming against the binary's *actual bytes*. On a non-Linux host, Unicorn plus a handful of PLT stubs is enough to run `main` for real — including the `mmap` / `mprotect` / `call` of the decrypted stub (`emu.py`):

```
$ python3 solve.py
[+] stub (38 bytes): 4889f849bbefbeaddea5a5a5a54c31d848c1c01149bbb3010000000100004 90fafc348f7d0
[+] stub constants: K=0xa5a5a5a5deadbeef M=0x100000001b3
[+] key: 0f6880178d2f3828  (8 raw bytes, non-printable)
[+] flag: HTB{WHY_S00_345Y_H0N}

$ python3 emu.py            # unicorn-backed, runs the binary's real code
input  : 0f6880178d2f3828
output : H4D5_UP_Y0U_M4D3_IT
         HTB{WHY_S00_345Y_H0N}

$ ./secondhand < key.bin    # on x86-64 Linux
H4D5_UP_Y0U_M4D3_IT
HTB{WHY_S00_345Y_H0N}
```

Emulating every input class confirms the branch map:

| input | length | result |
|---|---|---|
| `0f6880178d2f3828` | 8 | **flag** |
| `AAAAAAAA` | 8 | rough-cut narrative |
| `HTB{secondhand_rough_cut_v3}` | 28 | "…*It is not in the film.*" |
| `cda50dba01b2cb5578ce21798832f03b` | 16 | "…*It is not in the film.*" |
| anything else | any | rough-cut narrative |

## 7. Takeaways

- **Follow the length gates, not the symbol names.** `b_roll` and `threat_level_midnight` are the only two exported symbols, and both are dead ends. The real check is an unnamed static behind `len == 8`.
- **Distrust linear disassembly on a computed jump.** `add rdx, r11` before `jmp rdx` means objdump's instruction boundaries are a guess. Any large `movabs` immediate is a candidate hiding place — `0x1337c0debabeba49` and `0xb8491337c0debabe` are the same bytes at a one-byte offset, which is exactly the point.
- **Effort is not evidence.** The hand-rolled AES is the most sophisticated thing in the binary and it is worth nothing. Cheap wins — an XOR/rotate/multiply chain with an odd multiplier — invert in three lines and are where the flag actually lives.
- **A decoy in the right format is the strongest misdirection there is.** `HTB{secondhand_rough_cut_v3}` is well-formed, thematically on-point, and wrong; the binary politely tells you so if you ask it.