rev_dudsat — Doppler-Indexed Permutation Leak

HackTheBox Reversing · satellite Link Budget Processor stego · LCG-shuffled permutation table indexed by Doppler-themed FP arithmetic · flag delivered via a dead store the printed output never reads

Executive Summary

A 31 KB stripped x86-64 ELF impersonating a satellite Link Budget Processor (*lbproc v4.1.2*). Given a 1248-byte telemetry file `comms.dat` (26 records × 48 bytes per record), it prints a benign LOCK/NO-LOCK contact-window summary and exits cleanly. The flag is steganographically encoded in the printed table itself, but the bytes that carry each flag character are stored at `[rsp+7]` and never explicitly emitted — a classic dead-store-not-dead pattern. Recovery requires three steps:

(1) Reverse a Knuth / Numerical Recipes LCG-shuffle that runs in `.init_array[1]` and populates a 256-byte permutation table at `0x4040C0` before `main` executes. **(2)** Reverse a Doppler-themed velocity calculation per record that produces an index into that table. **(3)** Recognize that the lookup byte — discarded by the printed output — *is* the flag character for that record.

Flag: `HTB{d0pp13r_p3rm_134k_h7b}`

Reconnaissance

```
$ file lbproc
lbproc: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
  dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, stripped

$ ./lbproc comms.dat
LINK BUDGET PROCESSOR v4.1.2
Telemetry: comms.dat (26 records, 48 B/rec)
WINDOW  ID  STATUS
  001  003  LOCK
  002  007  LOCK
  ...
  024  041  NO-LOCK
  025  019  LOCK
  026  028  LOCK
Sessions: 24 LOCK, 2 NO-LOCK. Exit OK.
```

Nothing flag-shaped in stdout. `strings -n 6 lbproc` yields the version banner, printf templates, and PLT-stub names — no flag-shaped bytes. `comms.dat` is high-entropy (7.92 bits/byte) and looks like genuine FP/integer telemetry, not encrypted text. The challenge has to be in the code path that consumes the records.

`readelf -d lbproc` shows the tell: `INIT_ARRAY` with **two** entries. A stock C program has one (the linker's `_init` thunk). A second function in `.init_array` is something the developer added intentionally to run before `main` — and it is unreachable via normal call-graph exploration starting from `main`. That is layer one.

Layer 1 — the .init_array[1] constructor at 0x4011B0

Two functions matter here. A wrapper at `0x4011A0` first initializes the table to identity, then jumps into the shuffle at `0x4011B0`:

```
0x4011a0:  lea    rdi, [rip+0x2f19]             ; -> 0x4040C0 (.bss table)
0x4011a7:  mov    ecx, 256
0x4011ac:  xor    eax, eax
init_loop:
0x4011ae:  mov    byte ptr [rdi+rax], al        ; table[i] = i
0x4011b1:  inc    rax
0x4011b4:  cmp    rax, rcx
0x4011b7:  jl     init_loop
```

```
0x4011b9:  jmp    0x4011b0                      ; fall through to shuffle

0x4011b0:  push   rbp
0x4011b1:  mov    esi, 0x20FC8                  ; seed (deterministic!)
0x4011b6:  mov    rbp, rsp
0x4011b9:  push   rbx
0x4011ba:  mov    ebx, 0xff                     ; i = 255
0x4011bf:  sub    rsp, 0x8
0x4011c3:  lea    rdi, [rip+0x2ef6]             ; -> 0x4040C0
shuffle_top:
0x4011cd:  imul   esi, esi, 0x19660D            ; x = x * 0x19660D
0x4011d3:  lea    edx, [rbx+1]                  ; range = i+1
0x4011d6:  add    esi, 0x3C6EF35F               ;       + 0x3C6EF35F
0x4011dc:  mov    eax, esi
0x4011de:  mul    edx                           ; mulhi: rax:rdx = x*range
0x4011e0:  shr    rax, 0x20                     ; idx = (x*range) >> 32
0x4011e4:  mov    cl, [rdi+rbx]                 ; tmp        = table[i]
0x4011e7:  mov    dl, [rdi+rax]                 ; table[i]   = table[idx]
0x4011ea:  mov    [rdi+rbx], dl
0x4011ed:  mov    [rdi+rax], cl                 ; table[idx] = tmp
0x4011f0:  sub    rbx, 1
0x4011f4:  cmp    rbx, 0xFFFFFFFFFFFFFFFF
0x4011f8:  jne    shuffle_top
```

Three structural signals make the PRNG unambiguous:

(1) `imul`/`add` pair with constants `0x19660D` and `0x3C6EF35F` → Numerical Recipes *ranqd1* LCG (also Borland C `rand()`'s underlying generator). These constants are uncommon enough that the pair is fingerprintable on its own; revsolve v0.21.0 names both. **(2)** `mul edx` followed by `shr rax, 32` is the "scaled" modulo replacement — `(rand() * range) >> 32` — which avoids the modulo-bias problem and signals deliberate construction. **(3)** The seed `0x20FC8` is baked in: deterministic, reproducible, exactly what you want from a challenge author who needs the same permutation across every run.

The shuffle is textbook Fisher-Yates over a 256-element array, reproducible exactly:

```
def build_table():
    t = list(range(256))
    seed = 0x20FC8
    for i in range(255, -1, -1):
        seed = ((seed * 0x19660D) + 0x3C6EF35F) & 0xFFFFFFFF
        idx  = (seed * (i + 1)) >> 32
        t[i], t[idx] = t[idx], t[i]
    return bytes(t)
```

This recovers the exact 256-byte permutation that the binary builds in `.bss:0x4040C0` — verified by setting a hardware watchpoint on the table after the constructor returns and diffing against the Python output.

Layer 2 — Doppler-shifted index in main

After triaging the LOCK/NO-LOCK print loop, the floating-point work per record lives at `0x40123B–0x4012F0`. The relevant slice:

```
; per-record code, rsi -> comms.dat[record_n * 48]
0x40124a:  movsd     xmm0, qword [rsi+0x10]      ; b16 = freq_observed (Hz)
0x40124f:  movsd     xmm1, qword [rsi+0x18]      ; b24 = freq_emitted  (Hz)
0x401254:  movsd     xmm3, qword [rip+0xabc]     ; ~ 4.18e8  (K)
0x40125c:  movsd     xmm4, qword [rip+0xac0]     ; ~ 2.998e8 (c)
0x401264:  divsd     xmm0, xmm3                  ; xmm0 = b16 / K
0x401268:  addsd     xmm0, qword [rip+0xacc]     ; xmm0 += 1.0
0x401270:  mulsd     xmm0, qword [rsi+0x08]      ; xmm0 *= b8 (velocity)
0x401275:  subsd     xmm1, xmm0                  ; xmm2 = b24 - (b16/K+1)*b8
0x40127a:  cvttsd2si edx, xmm1                   ; idx = (int)xmm2
0x40127e:  and       edx, 0xff                   ; idx &= 0xFF
0x401283:  lea       rcx, [rip+0x2e36]           ; -> 0x4040C0 (table)
0x40128a:  mov       al, byte ptr [rcx+rdx]      ; observed = table[idx]
```

```
0x40128e:  mov       byte ptr [rsp+0x7], al      ; STORE on stack ...
; ... LOCK/NO-LOCK print follows, never reads [rsp+0x7] again
```

Two FP constants worth identifying explicitly:

Constant (hex qword)	Decimal	Identity
0x41B17DE83A000000	2.99792458×10^8	Speed of light in m/s — physical constant
0x41A8F5C1F1C00000	4.18229116×10^8	Misdirection: no physical meaning; ratio to c ≈ 1.395

The challenge is themed as a satellite link budget computation, and a genuine Doppler shift formula uses `c`. Here the author swapped in a *similar-looking* magic number so the calculation appears physics-y, but is actually just a deterministic byte-index computation: `(b16/K + 1.0) × b8`. The `c` constant is loaded but never used, purely to draw the eye.

The recovered formula:

```
idx                = int(b24 - (b16 / 418229116.0 + 1.0) * b8) & 0xFF
flag[record_n]     = permutation_table[idx]
```

The store at `[rsp+0x7]` is loaded but never read. In source the C likely was:

```
char observed = perm[idx];                  /* dead from this point */
log_status(record_n, parse_id(record));     /* prints LOCK/NO-LOCK */
```

The compiler kept the load and store because `perm[idx]` accesses memory and could trap, but the result lives in a stack slot that's discarded at function exit. Standard “dead store not dead” pattern — the *computation* is dead from the perspective of the program's visible behavior, but the *value* is the flag.

Solve

```
import struct

def build_table():
    t = list(range(256))
    seed = 0x20FC8
    for i in range(255, -1, -1):
        seed = ((seed * 0x19660D) + 0x3C6EF35F) & 0xFFFFFFFF
        idx  = (seed * (i + 1)) >> 32
        t[i], t[idx] = t[idx], t[i]
    return bytes(t)

data  = open('comms.dat', 'rb').read()
table = build_table()
out   = []
for n in range(26):
    rec = data[n*48 : (n+1)*48]
    b8  = struct.unpack('<d', rec[0x08:0x10])[0]
    b16 = struct.unpack('<d', rec[0x10:0x18])[0]
    b24 = struct.unpack('<d', rec[0x18:0x20])[0]
    idx = int(b24 - (b16/418229116.0 + 1.0)*b8) & 0xFF
    out.append(table[idx])
print(bytes(out))
# b'HTB{d0ppl3r_p3rm_l34k_h7b}'
```

26 records, 26 flag bytes — exact length match for an HTB flag. `comms.dat` was hand-crafted such that each record's `(b8, b16, b24)` triple produces the index whose corresponding permutation entry is the desired flag byte. Working backwards from the flag, the challenge author chose `b24` for each record to land on the right index. The FP-truncation step (`cvttsd2si`) is what makes that construction stable across compiler versions and rounding modes.

Why this earned its rating

Three layers of misdirection, each individually trivial but compounding:

(1) Two `.init_array` entries. Just uncommon enough to slip past anyone who doesn't `readelf -d`. Most analysts start at `main` and follow the call graph forward; the shuffle constructor sits outside that graph entirely.

(2) Doppler-themed FP constants. The constants make the index computation look like genuine physics modelling. Once you recognize that 4.18×10^{8} isn't `c` — and that `c` is loaded but never used — the misdirection collapses into an arithmetic byte-index.

(3) The dead store. Every static-analysis pipeline that prunes unused values will drop the `mov [rsp+0x7], al` and the table-lookup that feeds it, hiding the entire flag-extraction logic. The signal is in the load, not the store. Dynamic analysis with a hardware watchpoint on the permutation table reveals everything in one trace.

Difficulty earned through composition, not individual layer complexity. None of the pieces is hard in isolation; the trick is knowing where to look.

Artifacts & takeaways

Toolchain. As of revsolve v0.21.0, the fingerprint registry names both `0x19660D` (NR ranqd1 multiplier) and `0x3C6EF35F` (NR ranqd1 increment). The shuffle constructor is structurally recognizable in `auto` output without manual disassembly.

Patterns to remember.

• `imul reg, reg, 32bit_const` followed by `add reg, 32bit_const` within ~10 instructions of each other is an LCG. The specific constants identify the variant (Numerical Recipes ranqd1, glibc `rand()`, Java `Random`, etc.).

• `mul edx` / `shr rax, 32` after a PRNG step is a scaled-range index that avoids modulo bias. The presence of this idiom is itself a signal of careful construction.

• Dead stores fed by symbolic-execution-significant data are a recurring CTF trick. Always check what computation produces the dropped value before pruning. The store address (e.g. `[rsp+0x7]`) is your hook for a watchpoint.

• FP constants that *almost* match physical quantities are a misdirection signature. Compare every loaded FP qword against a small table of known physics/math constants (`c`, `e`, π, golden ratio, `k_B`) before assuming the math means what it claims.

Reproducibility. Given a fresh copy of `lbproc` and `comms.dat`, the solve above runs in < 50 ms in pure Python. No emulation, no debugger required.