Enthiran

HackTheBox Reverse Engineering — Insane

Author:	Cody Richard (aka ssstrickys)
Affiliation:	Independent Security Researcher / Global Thought Leader
Date:	May 15, 2026
Category:	Reverse Engineering (Neural-Network-Backed Binary)
Difficulty:	Insane
Flag:	HTB{n3ur4l_r3v3rs3r_1337}

Executive Summary

Enthiran is a stripped Linux ELF64 that presents itself as a benign system diagnostics utility but embeds a 16-32-8-1 multilayer perceptron whose hidden representation is consumed (in dead code) by a one-way encoder that emits the challenge flag. Conventional reversing — string analysis, control-flow exploration, comparison hunting — yields nothing usable, because no executed path inside the binary produces or reveals the flag. The puzzle is solved only by reasoning about the *mathematics of the model's internal representation*.

The network is trained so that, at a specific design input, its L2 hidden layer produces eight values that are exactly multiples of `1/256`. Floating-point rounding in the live forward pass corrupts the low byte of those activations, so naively re-running the network and dispatching the dead-code function yields garbage. Snapping each L2 output back to its exact `n/256` quantization point recovers the bit-perfect input the author used at build time, decoding the flag.

Final flag: `HTB{n3ur4l_r3v3rs3r_1337}`

1. Reconnaissance

1.1 Initial Triage

The supplied archive contained a single binary, `enthiran`, identifying as:

```
$ file enthiran
enthiran: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
          dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
          BuildID[sha1]=fa6db7a23e9a727e4f0120775ae8c8a130cddee3,
          for GNU/Linux 3.2.0, stripped
```

Executed normally, it produces the cover output expected of a diagnostic tool — a Linux banner, hardware readouts, an environment evaluation score, and an eight-element "diagnostic profile":

```
Enthiran System Diagnostics v1.0
================================
System: Linux 6.18.5 (x86_64)
CPUs:   1     RAM: 4003 MB     Uptime: 36 seconds
...
[*] Environment evaluation: score=0.991895
[*] Diagnostic profile:
    channel[0] = 1.2325344807
    ...
    channel[7] = 2.5362581901
[+] Profile analysis complete. Nominal.
```

The eight "channel" values fluctuate slightly between runs but cluster around stable means — the signature of a neural-network output whose inputs partially derive from environment entropy.

1.2 Import Surface

The dynamic symbol table immediately telegraphs the design:

```
exp, log2, sin, fmod                <- activation / numeric postprocessing
open, read, fopen, fscanf           <- /proc and /dev/urandom ingestion
clock_gettime, uname, sysconf       <- environment probing
putchar                             <- byte-level emission (suspicious)
```

Notably absent: `strcmp`, `memcmp`, or any single function used to match a flag string. The output pipeline relies on `__printf_chk` and `puts` for the diagnostic UI, leaving `putchar` as a stand-alone import — anomalous enough to flag for later scrutiny.

2. Mapping the Embedded Model

`.rodata` is 7,632 bytes — far more than a tool this small should need. Scanning for plausible IEEE-754 doubles revealed a contiguous block of 817 weights between `0x3310` and `0x4ca0`, with magic constants and threshold vectors after that point.

2.1 Layer Inventory

Each layer is computed by a single shared GEMV (general matrix-vector multiply) kernel at `0x1ed0`, parameterized with input and output dimensions and a stack-passed ReLU flag. Three call sites in `main` confirmed a clean three-layer architecture:

Layer	Dimensions	Weights addr	Bias addr	Activation
L1	16 → 32	0x3ca0	0x3ba0	ReLU
L2	32 → 8	0x33a0	0x3360	ReLU
L3	8 → 1	0x3320	0x3310	Sigmoid (score)

The eight values displayed to the user as "channels" are the L2 hidden representation. The reported *score* is `sigmoid(L3(L2))`, used solely to choose between two decoy print branches.

2.2 GEMV Kernel

The kernel itself is unremarkable but worth annotating because the weight matrix layout matters for any clean-room re-implementation. Weights are stored row-major as `W[in_idx * out_dim + out_idx]`:

```
out[o] = bias[o]
for i in range(in_dim):
    out[o] += input[i] * W[i * out_dim + o]
if relu_flag and out[o] <= 0:
    out[o] = 0.0
```

3. The Dead-Code Flag Emitter

Static analysis revealed a function at `0x2210` with **zero callers**. This was confirmed by three independent checks:

- No `e8` … direct call instruction anywhere in `.text` resolves to `0x2210`.
- No `lea` instruction loads its address into any register.
- No data-section qword or dword contains the value `0x2210`.

The function takes two arguments — `rdi` (pointer to eight input doubles) and `rsi` (output buffer) — and writes exactly 25 bytes. It is, structurally, the flag emitter.

3.1 Stage 1 — Prefix XOR

The first eight output bytes are produced by truncating each input double scaled by 256 to a 32-bit signed integer, then XORing the low byte with the corresponding byte of the constant at `0x3308`:

```
2240:  movsd     (%rdi,%rax,8), %xmm0   ; xmm0 = input[i]
2245:  mulsd     %xmm1, %xmm0           ; xmm0 *= 256.0
2249:  cvttsd2si %xmm0, %edx            ; edx  = int32(xmm0)  (truncate)
224d:  xor       (%rsi,%rax,1), %dl     ; dl  ^= key[i]
2250:  mov       %dl, (%rcx,%rax,1)     ; out[i] = dl
```

Key at `0x3308`: `DE AD BE EF CA FE BA BE` — the classic `0xDEADBEEFCAFEBABE` easter egg.

3.2 Stage 2 — 512-bit Hash Mixer

All 64 bytes of the input (eight little-endian doubles) feed into eight rounds of a SipHash-flavored mixer using well-known cryptographic constants:

- `0x736f6d6570736575` = ASCII "*somepseu*" — the standard SipHash v0 IV.
- `0x9e3779b97f4a7c15` — the golden-ratio constant used in xxHash and SplitMix64.
- `0xff51afd7ed558ccd` — the MurmurHash3 finalizer multiplier.

The result is a 64-bit state deterministically seeded by the bit-exact representation of the eight input doubles. There is no tractable inverse.

3.3 Stage 3 — SplitMix64 PRNG & Pad XOR

A SplitMix64-style PRNG keyed by the hash state runs *exactly* seventeen iterations (verified by computing `R_END / R_INC = 17` exactly under modular arithmetic), emitting one byte per iteration. Each output byte is XORed with a 17-byte pad at `0x32c0`:

```
Pad @ 0x32c0:
  33 d5 f5 55 07 f8 45 17 d0 7e 23 27 4e 3c 79 ef 78
```

Total output: 8 prefix bytes + 17 PRNG-XORed bytes + null terminator = 26 bytes (25 visible).

3.4 Build-Time Inversion

The encoder is one-way only in the sense that, given an unknown input, the output is unrecoverable. But the author used it *forwards*: they chose 8 doubles, computed the PRNG stream they produced, and stored `pad[i] = prng[i] ⊕ flag_byte[i+8]`. Running the function again with the same input round-trips the flag. The reverser's task is therefore to recover the author's 8 doubles.

4. Clean-Room Verification

Before tackling input discovery, the decoder was reimplemented in pure Python. Verification used a controlled experiment:

1. Set a GDB breakpoint at `0x1be3` — the instruction immediately following the L2 GEMV call.
2. Read the 8 doubles at `%rbp` (the L2 output buffer).
3. Invoke the binary's own `0x2210` function in-process via `(void(*)(double*,char*))` with those 8 doubles.
4. Compare the 25-byte output against the Python re-implementation.

The byte-for-byte match (`e78984edda072b3d6c43...`) confirmed semantic equivalence and provided a fast Python oracle for the search ahead.

5. Discovering the Design Input

5.1 Structure of the L1 Input Vector

Tracing the construction of the 16-dimensional L1 input vector established the following layout on the stack at `0xd0(%rsp)`:

- **Positions 0–8 (nine values):** blended as `0.65 · env_signal + 0.35 · constant`, with constants `[0.61, 0.55, 0.78, 0.29, 0.44, 0.67, 0.52, 0.38, 0.66]` from `.rodata`.
- **Position 10:** a single environment-derived scalar.
- **Positions 9, 11–15:** six hardcoded constants `[0.72, 0.41, 0.85, 0.33, 0.91, 0.23]` — not environment-dependent.

Critically, if the environment-derived signals at positions 0–8 happen to equal their paired constants, the blend collapses to the constants themselves, making the L1 vector almost entirely deterministic save for position 10.

5.2 Sweeping the One Free Dimension

With positions 0–8 pinned to their constants and 9, 11–15 left at their hardcoded values, position 10 was swept from 0 to 1. At `pos10 = 0.5`, the L2 output revealed its structure:

```
ch[0] = 0.5859374999999993   ~ 150 / 256   (delta ~ 6.7e-16)
ch[1] = 0.9726562499999998   ~ 249 / 256   (delta ~ 2.2e-16)
ch[2] = 0.9843750000000008   ~ 252 / 256   (delta ~ 7.8e-16)
ch[3] = 0.578125000000000    = 148 / 256   (exact)
ch[4] = 0.6406250000000003   ~ 164 / 256   (delta ~ 3.3e-16)
ch[5] = 0.8007812499999993   ~ 205 / 256   (delta ~ 6.7e-16)
ch[6] = 0.8085937499999998   ~ 207 / 256   (delta ~ 2.2e-16)
ch[7] = 0.7968749999999990   ~ 204 / 256   (delta ~ 1.0e-15)
```

Every output channel falls within ~10^{-15} of an exact `n/256`. The network was trained so its hidden representation lands precisely on a quantization grid that matches the encoder's `int(x · 256)` step.

5.3 The Rounding Trap

The deltas are tiny, but they are *just* enough to flip the low byte after truncation. For example, `int(0.5859374999999993 · 256) = 149` rather than 150 — producing prefix `K U B {` instead of `H T B {`.

More importantly, the stage-2 hash mixer is sensitive to *all 64 bits* of every input double. Even if the prefix accidentally landed correctly, the suffix would still be garbage because the rounding error propagates through the hash. The fix had to operate on the bit pattern of each input.

5.4 Snapping to the Grid

Rounding each L2 output to its nearest `n/256` and feeding the resulting eight doubles back through the decoder:

```
>>> target_L2 = [150/256, 249/256, 252/256, 148/256,
                 164/256, 205/256, 207/256, 204/256]
>>> emit_flag(target_L2)
b'HTB{n3ur4l_r3v3rs3r_1337}'
```

Decoded flag: `HTB{n3ur4l_r3v3rs3r_1337}`

6. Why This Earned an "Insane" Rating

Every standard reverse-engineering technique fails in isolation:

- There is no comparison instruction to defeat — nothing in the binary checks user input against a stored key.
- There is no executed control-flow path that produces the flag — the flag emitter at `0x2210` is genuinely dead code.
- Brute-forcing the eight input doubles is infeasible: the SplitMix64 stage depends on all 64 input bits across eight 64-bit values (effectively a 512-bit search space).

- Naively executing the network and dispatching the dead function produces garbage on every machine, because the live L2 output is rounded off the design grid.
- The two visible decision branches (“Nominal” vs “No anomalies detected”) are decoys: neither reaches the flag emitter.

The intended solution explicitly demands the analyst *understand* the model — not just its computation graph, but the geometric structure of its trained activations. The hidden representation, when interpreted with full numerical awareness, is itself the key.

7. Artifacts & Takeaways

7.1 Artifacts Produced

- `flagdec.py` — clean-room re-implementation of the encoder at `0x2210`, byte-perfect against the binary.
- GDB scripts for breakpoints at the L1 input (`0x19f4`) and L2 output (`0x1be3`) sites, including an in-process invocation of `0x2210` with a custom `rdi/rsi` setup.
- Pure-Python forward pass of the embedded MLP (matches the binary’s output bit-for-bit with the same input vector).

7.2 Defensive & Offensive Lessons

- **Steganographic neural networks are a real attack-surface concept.** A trained model whose activations carry hidden payloads can ship as innocuous-looking weights in any binary, library, or model artifact.
- **Dead code is a legitimate carrier.** An orphan function with no callers can encode meaningful semantics that only an analyst can discover — useful both for puzzle design and for implant/payload concealment.
- **Floating-point quantization grids deserve attention.** When model activations consistently land near rational fractions, suspect a deliberate alignment with downstream byte-level operations.
- **Trust dynamic analysis only to the bit level.** Reading “close enough” values from a live process can mask the intended design point.

Writeup by Cody Richard (ssstrickys) · Independent Security Researcher · May 15, 2026