# rev_sysprobe — Five-Layer Packer to a Bit-Encoded Flag

HackTheBox Reversing · hidden RWX PT_LOAD → self-decompressing loader → section-stripped inner ELF → index-keyed-XOR bytecode → 2-bits-per-slot rodata table · end-to-end recovery in two commands

## Executive Summary

A 33 KB x86-64 ELF presenting as a system diagnostics tool (*sysprobe v1.4.2*). Behaviorally indistinguishable from a benign sysadmin utility until you read its strings — at which point you find C2 callouts and systemd persistence vectors. The flag lives behind **five distinct layers**, each individually approachable, sequenced to punish anyone without the right tooling at each gate:

| # | Layer | Technique required |
|---|---|---|
| 1 | Hidden RWX PT_LOAD at `0x804000` | `readelf -l` — check program headers, not sections |
| 2 | Self-decompressing loader (`mmap` + `call rbx`) | Dynamic analysis — gdb-driven runtime memory dump |
| 3 | Inner ELF with no section headers | PT_LOAD-only ELF parser fallback |
| 4 | VM bytecode with index-keyed XOR | `deobf[i] = obf[i] ^ ((0x42+i) & 0xFF)` |
| 5 | Flag in `QB_REAL`, 2 bits/slot encoding | 128 slots × 8 B — bit-decode under permutation search |

**Flag:** `HTB{TH15_TH3_END_0R_WH4T}`

With current tooling (revsolve v0.21.0), end-to-end recovery is two commands: `revsolve runtime-dump sysprobe` followed by an automatic auto-pivot that prints the flag inline. Before that tooling existed, this challenge required ~seven hand-rolled gdb scripts to extract the decompressed inner image plus another round of analysis to recognize `QB_REAL` as a bit-encoded blob.

## Reconnaissance

```
$ file sysprobe
sysprobe: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
  dynamically linked, stripped

$ ./sysprobe
[sysprobe] system diagnostics v1.4.2
[sysprobe] checking /proc/cpuinfo ...     [OK]
[sysprobe] checking systemd units ...     [OK]
[sysprobe] enumerating cron tasks ...     [OK]
[sysprobe] beacon active
[sysprobe] complete (exit 0)
```

Nothing flag-shaped on stdout. Strings tell a different story:

```
$ strings -a sysprobe | grep -iE 'flag|HTB|cron|systemd|beacon'
update-monitor.net
/etc/systemd/system/sysprobe.service
/etc/cron.d/sysprobe
[vm] persistence written: /etc/systemd/system/sysprobe.service
[vm] persistence attempt: /etc/cron.d/sysprobe
SYSPROBE_BEACON_TS=
```

Malware-themed. The `[vm]` prefix is the tell: there's an embedded VM somewhere. `update-monitor.net` is a sandboxed C2 placeholder (no DNS in HTB infra). No HTB-shaped string anywhere — the flag is not in the outer's strings.

`readelf -l sysprobe` exposes the structural anomaly:

```
Program Headers:
  Type    Offset    VirtAddr           FileSiz   MemSiz    Flg Align
  PHDR    ...
  INTERP  ...
  LOAD    0x000000  0x0000000000400000 0x002a40  0x002a40  R E 0x1000
  LOAD    0x003000  0x0000000000403000 0x000200  0x000200  RW  0x1000
  LOAD    0x004000  0x0000000000804000 0x001fa0  0x001fa0  RWE 0x1000  <--
  DYNAMIC ...
```

**PT_LOAD #3 is RWX**, mapped at `0x804000`, 8096 bytes long, sourced from file offset `0x4000`. No section maps to this range — `readelf -S` confirms `.bss` ends at `0x4034C8` and there are no more sections. This is a sleeper segment: the program will copy it into freshly-mmaped RWX memory and call into it. `e_entry = 0x804a55` — *inside* the hidden segment — confirms it.

## Layer 1 to 2 — the decompression stub

Static disassembly of the hidden segment starting at the entry point:

```
0x804a55:  endbr64
0x804a59:  push     rbp
0x804a5a:  mov      rbp, rsp
0x804a5d:  sub      rsp, 0x10
0x804a61:  mov      edi, 0                ; addr  = NULL
0x804a66:  mov      esi, 9464             ; size  = 9464
0x804a6b:  mov      edx, 7                ; prot  = PROT_RWX
0x804a70:  mov      ecx, 0x22             ; flags = MAP_PRIVATE|MAP_ANON
0x804a75:  mov      r8d, -1               ; fd    = -1
0x804a7b:  xor      r9d, r9d              ; off   = 0
0x804a7e:  call     mmap@plt
0x804a83:  mov      rbx, rax              ; rbx = decompressed image base
0x804a86:  lea      rdi, [rip+...]        ; src = compressed @ 0x804100
0x804a8d:  mov      rsi, rbx              ; dst = mmaped region
0x804a90:  call     0x8045b5              ; decompress(src, dst)
0x804a95:  call     rbx                   ; <-- jump into decompressed code
0x804a97:  ...                            ; cleanup, exit
```

This is the canonical packer signature: `PROT_RWX` + `MAP_ANON` + decompress-then-`call rbx`. Static analysis stops here. The decompressor at `0x8045B5` is a custom bit-reader (variable-length encoding, single-byte literals interleaved with reference lengths) — not LZ4, not zlib, not LZMA. Reversing it statically would take an afternoon. Reversing it dynamically takes one minute.

## Layer 3 — runtime extraction

The right approach is to let the loader do its work and snapshot the decompressed memory just before `call rbx` fires. With revsolve v0.21.0:

```
$ revsolve runtime-dump sysprobe
# revsolve runtime-dump
- Target:    sysprobe
- Captures:  **1**

| Base addr      | Size  | Prot | Path                  | Auto-pivot |
| 0x7ffff7fb8000 | 12288 | RWX  | sysprobe.runtime/...  | top archetype
  `embedded_classifier` @ 0.90; recovered blob (1.00 printable):
  `HTB{TH15_TH3_END_0R_WH4T}}`; 1 orphan function(s)
```

Under the hood:

• gdb is launched with the binary, `catch syscall mmap` is set, and a Python `gdb.events.stop` listener filters on `$orig_rax == 9` (sys_mmap on x86-64). • The listener implements a two-state machine: on syscall entry with `rdx == 7` (PROT_RWX) and `min ≤ rsi ≤ max`, it arms. On the matching syscall exit it reads `rax` (the returned address) and installs a transient breakpoint at that address. • When that breakpoint fires — which it does exactly when the loader executes its first instruction inside the new region, *after* decompression — the listener walks `info proc mappings` to find the containing RWX range and dumps the entire page-aligned mapping via `dump binary memory`.

One last detail: many self-decompressing loaders zero the leading `0x7F` ELF-magic byte of their decompressed inner ELFs as a static-analysis tripwire. `file` rejects the dump; pyelftools rejects the dump; anything that checks the magic rejects the dump. revsolve checks bytes 1–3 against `ELF` and the rest of the e_ident against sane values, and if everything else looks like a real ELF, it patches the magic back. The dump is then a parseable ELF for downstream consumers without special-casing.

Result: a 12288-byte inner ELF image. `strings -a` on the dump reveals the symbol table the inner kept:

```
payload_entry
vm_run
_binary_vm_bytecode_bin_start
_binary_vm_bytecode_bin_end
_binary_bytecode_vm_bytecode_obf_bin_size
QB_REAL
```

Two free hints in that list. First, `vm_run` implies a stack-VM interpreter — reverse it once and the rest is data decoding. Second, the `_obf` suffix in `_binary_bytecode_vm_bytecode_obf_bin_size` is the author saying out loud: the bytecode *is* obfuscated.

## Layer 4 — deobfuscating the VM bytecode

The inner ELF's section headers are stripped — pyelftools raises `ELFParseError: missing section headers at 6040`. revsolve's PT-only fallback walks the program headers and synthesizes `.pt_text`, `.pt_rodata`, and `.pt_data` records. The 62-byte obfuscated bytecode blob lives in `.pt_data:0x1ca0`.

**Recovery via live diff.** Set a breakpoint at `vm_run`, run, and inspect the first argument. By the amd64 SysV ABI, `rdi` on function entry holds the first pointer-typed argument — here, the bytecode buffer. The bytes pointed to by `rdi` are the live, deobfuscated bytecode the VM is about to execute. Diff against the on-disk obfuscated bytes:

```
(gdb) x/16bx $rdi
0x7ffff7fb9ca0:  0x01 0x00 0x06 0x02 0x0c ...     ; live (deobfuscated)
(gdb) x/16bx ((char*)inner_base + 0x1ca0)
0x7ffff7fb8ca0:  0x43 0x43 0x44 0x41 0x48 ...     ; disk (obfuscated)
```

Solve over the first few bytes by elimination: the difference is a position-dependent function of `i`. Trying constant XOR fails. Trying `obf[i] ^ (k + i)` for `k = 0x42, stride = 1` fits all observed pairs. Confirm with the full 62 bytes — clean decode.

```
deobf[i] = obf[i] ^ ((0x42 + i) & 0xFF)

obf  = open('inner_dump.bin','rb').read()[0x1ca0:0x1ca0+62]
deob = bytes(b ^ ((0x42 + i) & 0xFF) for i, b in enumerate(obf))
```

The deobfuscated bytecode disassembles cleanly under the VM ISA recovered from `vm_run`'s dispatch loop. But it does *not* contain the flag — it contains instructions that emit the `[sysprobe] beacon active` string, that attempt the systemd persistence write, and that exit with status 0. The bytecode is the malware-theater. The flag is elsewhere.

## Layer 5 — QB_REAL bit-decode

The `QB_REAL` symbol in the inner ELF points to a 1024-byte rodata table. Naive interpretation: an IEEE-754 lookup of 128 doubles. Statistical inspection refutes this:

```
$ python -c 'import struct,sys
data = open("inner_dump.bin","rb").read()
qb   = data[QB_REAL_OFFSET:QB_REAL_OFFSET+1024]
slots = set(qb[i:i+8] for i in range(0, 1024, 8))
print(len(slots), "distinct 8-byte slots")
for s in sorted(slots): print(s.hex())'

4 distinct 8-byte slots
0000000000000000
4035940158b04d06
58b04d06c0ca6bfe
99e5e107187bb904
```

1024 bytes / 8 bytes-per-slot = 128 slots, of which only 4 distinct values appear. As IEEE-754 doubles three of those bit patterns are garbage (denormals, near-zero magnitudes); they are not floats, they are 8-byte symbols. With 4 distinct symbols, each carries $\log_2(4) = 2$ bits of information. 128 slots × 2 bits = 256 bits = **32 bytes**. 32 ASCII bytes is the right shape for an HTB flag.

Two ambiguities remain. **(a)** Which symbol maps to which 2-bit value? Four symbols, 4! = 24 permutations. **(b)** Within each output byte, are the four 2-bit slots concatenated MSB-first or LSB-first? Two orderings. 48 total candidate decodings — trivially brute-forceable. Score each by the fraction of decoded bytes that are printable ASCII; keep the unique winner.

```
from itertools import permutations

VALUES = [bytes.fromhex(v) for v in (
    "0000000000000000", "4035940158b04d06",
    "58b04d06c0ca6bfe", "99e5e107187bb904")]

slots = [qb[i:i+8] for i in range(0, 1024, 8)]
syms  = [VALUES.index(s) for s in slots]

for perm in permutations(range(4)):
    for order in ("msb", "lsb"):
        bits = []
        for sym in syms:
            v = perm[sym]
            if order == "msb":
                bits.extend([(v >> 1) & 1, v & 1])
            else:
                bits.extend([v & 1, (v >> 1) & 1])
        out = bytes(
            int(''.join(str(b) for b in bits[i:i+8]), 2)
            for i in range(0, len(bits), 8))
        printable = sum(0x20 <= c < 0x7F for c in out) / len(out)
        if printable > 0.95:
            print(perm, order, out)
            break
```

Winning combination: permutation `(1, 0, 2, 3)`, MSB-first within each byte. Output:

```
(1, 0, 2, 3) msb b'HTB{TH15_TH3_END_0R_WH4T}}\x00\x00\x00\x00\x00\x00'
```

Trailing nulls are slot padding (the flag is 26 bytes; 32 - 26 = 6 trailing slots that decode to `0x00` bytes). The duplicate `}` at the end is a one-byte fence/sentinel the author added to make the boundary unambiguous. The flag is `HTB{TH15_TH3_END_0R_WH4T}`.

## Why this earned its rating ("Hard")

Composition over individual layer complexity. None of the five layers is hard in isolation:

• Hidden PT_LOAD is `readelf -l`. • `mmap` + `call rbx` is a debugger run. • Section-stripped ELF needs only PT_LOAD-aware tooling. • Index-keyed XOR yields to a one-window trial-and-error. • 4-distinct-value rodata table is 48

permutations of brute force.

Sequenced, the layers punish every tooling shortcut. Pre-v0.21.0 revsolve crashed at layer 3 because pyelftools rejects the section-stripped inner ELF. The previous fix (PT-only fallback) was implemented but its end-to-end path was not exercised. v0.21.0 adds `runtime-dump` for layer 2 plus the `encoded_blobs` auto-pivot wiring for layer 5. The rev_sysprobe end-to-end recovery served as the regression target for both fixes.

## Artifacts & takeaways

**Hidden RWX PT_LOAD with no section coverage is the canonical packer signature.** Always `readelf -l`, not just `readelf -S`. Anything mapped RWX in production code outside of JIT-heavy runtimes is suspicious; in CTF binaries it is always the lead.

**Section-stripped ELFs are increasingly common** in both CTFs and real malware. Tooling must handle PT_LOAD-only parsing or it stops at the boundary. The fallback parse is straightforward: categorize each PT_LOAD by its flags (`X` → `.pt_text`, `W` → `.pt_data`, neither → `.pt_rodata`) and consume downstream as if they were real sections.

**Index-keyed XOR (`obf[i] ^ (k + i)`) is the obfuscation choice** when the author wants something stronger than constant XOR but doesn't want a real cipher. It decodes cleanly for exactly one `(base, stride)` pair and rejects all others under any reasonable English-printability filter. The signature: a 62-byte blob whose first 16 bytes decode to a coherent ASCII run only for `k = 0x42, stride = 1`.

**Low-symbol-count tables** are the QB_REAL pattern. Specifically: an aligned rodata blob whose total length divides cleanly by 8 (or 4, or 2), with a small power-of-two count of distinct values. Compute $\log_2(\text{distinct})$, multiply by slot count, divide by 8 — if the byte count matches a plausible flag length (typically 20–40 bytes for HTB), bit-decode under all permutation × bit-order combinations. The brute force is cheap (48 candidates here; up to 384 for 8 distinct values). The win condition is unique: exactly one combination yields > 95% printable ASCII; everything else is < 30%.

**Symbol-name leakage matters.** The `_obf` suffix in `_binary_bytecode_vm_bytecode_obf_bin_size` is a CTF-author hint. Production packers strip the symbol table (`strip`, or `objcopy --strip-unneeded`), but CTFs frequently leave breadcrumbs to keep the challenge fair. Always read the remaining symbols before disassembling.

**End-to-end recovery with revsolve v0.21.0:**

```
$ revsolve runtime-dump sysprobe
# revsolve runtime-dump
- Captures: 1
0x7ffff7fb8000 | 12288 | RWX | sysprobe.runtime/dump_7ffff7fb8000.bin
  Auto-pivot: top archetype `embedded_classifier` @ 0.90;
    recovered blob (1.00 printable): `HTB{TH15_TH3_END_0R_WH4T}}`;
    1 orphan function(s)
```

Two commands turn a five-layer packer into a one-line flag print. The investment in tooling pays for itself the first time you encounter the next variation on this pattern.