HACKERDNA REVERSING · ARM64 LICENSE CRACKME

Sentinel License Manager

not-stripped aarch64 → XOR token leak → reversed `derive_key` → forged admin keygen

CHALLENGE	CATEGORY	DIFFICULTY
sentinel-activate	Reversing · Keygen	Medium
TARGET	RESULT	OPERATOR
aarch64 ELF (PIE)	both flags	Cody Richard · ssstrickys

TECHNICAL BRIEF · EXECUTIVE SUMMARY

A license check keyed on nothing but the account name

Category **Reversing** · Difficulty **Medium** · ARM64 static analysis + keygen

Sentinel ships a small ARM64 command-line activation client, `sentinel-activate`, that validates a `SENT-XXXX-XXXX-XXXX` license key against an account name. Premium modules are gated behind a valid **admin** license. The binary is **not stripped** — four symbols (`main`, `validate`, `derive_key`, `decode_token`) hand you the entire design. Two independent wins fall out: an **undocumented `--diag` subcommand** that XOR-decodes and prints a hidden flag, and a license routine that compares your key to a pure function of the account name — so reimplementing `derive_key` forges a valid key for **any** account, admin included.

OBJECTIVES

#	GOAL	TECHNIQUE
1	Recover the hidden user flag	Static XOR (0x5A) of a `.rodata` blob via `decode_token` / `--diag`
2	Forge the admin license	Reverse `derive_key` → Python keygen → premium flag

01 · TRIAGE

Pull the binary, read the symbols

The activation portal serves the client at `/download/sentinel-activate`; the box exposes only the web tier and an FTP port.

First look: an ARM64 ELF, dynamically linked, and — critically — **not stripped**. On an x86 host the stock `objdump` refuses aarch64 (`architecture UNKNOWN`), so all disassembly here is via **radare2**. The symbol table alone sketches the whole program.

```
$ FILE SENTINEL-ACTIVATE ; NM SENTINEL-ACTIVATE | GREP ' T '                TRIAGE

ELF 64-bit LSB executable, ARM aarch64 ... not stripped
0x00400744 t decode_token
0x004007b0 t derive_key
0x004009f8 t validate
0x00400ac4 t main
```

Strings confirm the shape of the problem — the key format, the success/failure lines, and a tell-tale `User Flag: %s`:

```
$ STRINGS -N6 SENTINEL-ACTIVATE                                             STRINGS

SENT-%c%c%c%c-%c%c%c%c-%c%c%c%c
User Flag: %s
VALID: license accepted for account '%s'
INVALID: license rejected for account '%s'
```

02 · THE HIDDEN FLAG — DECODE_TOKEN & THE --DIAG DOOR

A 36-byte XOR blob and an argument the usage never mentions

`main` parses three flags: `--account`, `--key`, and an undocumented `--diag`.

The `--diag` branch calls `decode_token` and prints its result through the `User Flag: %s` format — a diagnostic backdoor that never appears in the usage string. `decode_token` itself is trivial once disassembled: it walks a 36-byte blob in `.rodata` at `0x400cc0` and XORs every byte with **0x5A**.

RADARE2 · PDF @ SYM.DECODE_TOKEN 0X400744

```
0x00400758  add x1, x0, 0xcc0        ; .rodata blob "bc;<ln<mw..."
0x00400760  ldrb w2, [x1, x0]        ; c = blob[i]
0x00400770  mov  w1, 0x5a            ; key = 0x5A
0x00400774  eor  w1, w2, w1          ; out[i] = c ^ 0x5A
0x0040079c  ... strb wzr             ; NUL-terminate at [36]
```

Decoding statically — no need to even run the binary — yields the hidden user flag (a UUID). Redacted here:

USER FLAG **89af64f7-••••••••••••••••••••••••••** REDACTED

THE SHORTCUT

Because the blob is static, the flag is recoverable purely from `.rodata` — `python3 -c "print(''.join(chr(b^0x5A) for b in BLOB))"`. Running `./sentinel-activate --diag` under qemu prints the identical string.

03 · THE LICENSE CHECK — VALIDATE

Uppercase the key, strcmp against derive_key(account)

There is no server round-trip and no embedded secret — `validate` is offline and deterministic.

`validate(account, key)` does exactly two things: it upper-cases the user-supplied key in place (so keys are case-insensitive), then `strcmp`s it against the output of `derive_key(account)`. Match → return 1 (VALID); else 0 (INVALID). Whatever `derive_key` produces for a given account name **is** the valid key for that account.

RADARE2 · PDF @ SYM.VALIDATE 0X4009F8

```
0x00400a14  bl  sym.derive_key       ; expected = derive_key(account)
0x00400a50  sub w0, w0, 0x20         ; toupper() on each key char
0x00400aac  bl  sym.imp.strcmp       ; strcmp(expected, UPPER(key))
0x00400ab4  cset w0, eq              ; return (match)
```

04 · REVERSING DERIVE_KEY

A 6-byte rolling XOR digest over the account name

The heart of the crackme: a per-character transform folded into a six-byte state, then hex-encoded into the `SENT-` format.

For each character `c` of the account at index `i` (with `len = strlen(account)`), the routine computes a transformed byte and XORs it into `state[i % 6]`. One subtlety trips a naive port: the intermediate `a` is held in a 32-bit register and is **not** truncated to a byte before the next step — only `b` onward are byte-wide.

RECONSTRUCTED PER-CHAR TRANSFORM 0X400808

```
for i, c in enumerate(account):
    a = ((c & 0x7f) << 1) + c          # 32-bit, NOT byte-masked
    b = (a + ((a & 0x1f) << 3)) & 0xff  # strb -> byte
    d = (b + 0x3d) & 0xff
    e = d ^ 0x47
    g = (e + i) & 0xff
    h = (g ^ len) & 0xff
    state[i % 6] ^= h                   # rolling 6-byte state
```

After the fold, the six state bytes are emitted as twelve upper-case hex digits (table `"0123456789ABCDEF"` at `0x400e18`) and `sprintf`'d into `SENT-XXXX-XXXX-XXXX`.

ORACLE FOR FREE

The activation portal publishes a working **trial** credential — account `trial`, key `SENT-3B00-1C47-EF00`. That is a built-in unit test: any correct reimplementation of `derive_key("trial")` must reproduce that exact string.

05 · KEYGEN & FORGING ADMIN

Verify on the trial oracle, then mint the admin license

The Python port reproduces the trial key byte-for-byte, proving the algorithm is exact. Feeding it `admin` forges the administrator license; submitting that to the activation endpoint unlocks the premium tier and returns the second flag.

```
$ PYTHON3 SENTINEL_KEYGEN.PY ADMIN                                  KEYGEN

[+] hidden user flag (--diag / decode_token): 89af64f7-····-····-····-····
[+] valid key for account 'admin': SENT-3A8A-····-····
[i] self-test derive_key('trial') == SENT-3B00-1C47-EF00  ->  OK
```

```
$ CURL -S -X POST .../ACTIVATE -D 'ACCOUNT=ADMIN&LICENSE_KEY=SENT-3A8A-····-····'  ACTIVATE

{"account":"admin","tier":"premium",
 "message":"Premium license activated. All modules unlocked.",
 "root_flag":"f8694e5c-····-····-····-····","status":"success"}
```

PREMIUM FLAG `f8694e5c-····-····-····-····` REDACTED

06 · LESSONS

Why this design fails

Symmetric, offline validation. When the client both derives and checks the key, no network secret is involved — the whole trust model reduces to obscurity of a local function, which static analysis dissolves. A real activation scheme signs the license server-side (asymmetric) so the client can verify but never *mint*.

Ship-stripped, drop diagnostics. An un-stripped binary with named `derive_key`/`decode_token` symbols and a hidden `--diag` flag is a gift to an analyst. Diagnostic backdoors that print secrets must never survive to release.

Trivial obfuscation ≠ protection. A single-byte XOR (0x5A) over a static blob is decoration; it is recovered without ever executing the code.