CYBER APOCALYPSE 2026 · CRYPTO

Fractured Seal

Category: Crypto **Primitive:** RSA **Technique:** Coppersmith / partial key exposure **Solve time:** 5.7 s

---

FLAG

HTB{r3c0v3r1ng_RSA_k3ys___l1k3___Me0w___me0o00o0o0w___Me0w}

> *One of the Registry's oldest key-scrolls survived the fall of Crownspire, though time and fire spared only fragments of its writing, and most in the vault dismissed it as useless. Caldrin didn't. She always said a seal doesn't have to be whole to still remember the door it once opened.*

1. Files provided

File	Description
`encrypt.py`	The encryption script
`flag.enc`	256-byte raw RSA ciphertext
`fractured_seal.pem`	An RSA private key with most characters replaced by `*`

`encrypt.py` is textbook, unpadded RSA:

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import long_to_bytes, bytes_to_long, getPrime

p = getPrime(1024)
q = getPrime(1024)
n = p * q
e = 0x10001
d = pow(e, -1, (p-1)*(q-1))

m = bytes_to_long(open('flag.txt', 'rb').read())

open('seal.pem', 'wb').write(RSA.construct((n, e, d)).export_key())
open('flag.enc', 'wb').write(long_to_bytes(pow(m, e, n)))
```

No padding, no exotic maths, standard 2048-bit modulus and `e = 65537`. There is nothing wrong with the *scheme* — so the entire challenge lives in the damaged PEM file.

2. Looking at the damaged key

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAjK59ahXlX7a+oF+jt5icukpGeNXXgQO4D3jPeLaGAupJm6a6
9PnWCf0W3QDhmCPxLYWSr1C4DvxP23UlvP8Lcfu+w/oIS0jDHkfnv+m1Qku/ii9w
ehy/iRNuUeaH88K4AA0XFvJoak5I0Igi5ksP/CsyGMRJbUe898eLZjYOYAoA2+fi
LjjfYDBHliyXrva55W9O2ie2SNZ2Q859kAb0IQj/DK3LuAHqtuE7HVm2KDydU0jj
iMFz5uhwcU5njXnHdZRfrCTaUWBLLxmE8PCbIuy8hJqhL+iD1OcwTS8Yo8xoTtoc
7HFFfXYNoKjnqgp+LOmLwQGQeO2Yo1Hb8SCVd**************************
************************************************************
...
**************************************************2msCgYEAwLGx
cJ7/YCgq9GPDS16cHPNZEmYrbSX+atzUBBO2jLYg0QbXfitTIHfU+55DqIxFQOcu
+CahrPQQROoZAAIPg0LdaGd+3R3/ri*********************************
************************************************************
****************************    l
**************************    (ᐟ 。 7
**************************   l、 ~ロ
**************************   じしf_, )ノ
************************************************************
...
-----END RSA PRIVATE KEY-----
```

Two things matter here:

1. **The `*` characters are in-place substitutions, not deletions.** Every masked character occupies exactly one Base64 position, so the surviving characters are still at their original offsets. That is what makes recovery possible at all.
2. **Lines 18–21 of the body contain an ASCII-art cat that ate some of the `*`s.** Those lines are 30/27/29/29 Base64 characters long instead of 64. To keep the offsets of everything *after* them correct, each of those lines must be restored to 64 masked characters. (In this challenge the surviving fragments all sit *before* the cat, so this only affects the length sanity-check — but getting alignment right is what lets us trust the parse.)

After normalisation the body is **1588 Base64 characters = 397 groups = 1191 DER bytes**.

3. Recovering the surviving bytes

Base64 packs 3 bytes into 4 characters. Within one group of 4 characters `c0 c1 c2 c3`:

- byte 0 depends on `c0, c1`
- byte 1 depends on `c1, c2`
- byte 2 depends on `c2, c3`

So a byte is recoverable whenever **both** of the characters it depends on survived — a group with 3 known characters still yields 2 known bytes. Decoding at this granularity rather than group granularity gains one extra byte at the end of the second fragment, which turns out to be free precision.

```
import base64

raw = [l.rstrip('\n') for l in open('fractured_seal.pem')][1:-1]
B64 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

lines = [''.join(c for c in l if c in B64 + '*') for l in raw]
# lines 17..20 (0-indexed) are mangled by the ASCII cat -> restore full-width masks
s = ''.join(lines[:17]) + '*' * 256 + ''.join(lines[21:])
assert len(s) == 1588

known = {}
for g in range(len(s) // 4):
    v = [B64.index(c) if c != '*' else None for c in s[4*g:4*g+4]]
    if v[0] is not None and v[1] is not None: known[3*g]     = (v[0] << 2) | (v[1] >> 4)
    if v[1] is not None and v[2] is not None: known[3*g + 1] = ((v[1] & 15) << 4) | (v[2] >> 2)
    if v[2] is not None and v[3] is not None: known[3*g + 2] = ((v[2] & 3) << 6) | v[3]
```

This yields exactly two surviving runs of DER bytes:

```
known byte runs: [(0, 266), (663, 741)]
```

346 bytes out of 1191 — about 29% of the key.

4. Mapping the fragments onto the DER structure

A PKCS#1 `RSAPrivateKey` is a fixed-shape DER `SEQUENCE`:

```
RSAPrivateKey ::= SEQUENCE {
    version, modulus n, publicExponent e, privateExponent d,
    prime1 p, prime2 q, exponent1 dP, exponent2 dQ, coefficient qInv
}
```

Because every field length is predictable for a 2048-bit key, the byte offsets can be laid out exactly. The first surviving run begins with `30 82 04 a3 02 01 00 02 82 01 01 00 8c ae 7d ...` which confirms it:

Offset	Bytes	Field	Status
0–3	`30 82 04 a3`	SEQUENCE, 1187 bytes content	known (1187 + 4 = 1191 ✓)
4–6	`02 01 00`	`version = 0`	known
7–10	`02 82 01 01`	INTEGER, 257 bytes	known
11–267		**`n`** (leading `00`)	**known except the very last byte (offset 267)**
268–272	`02 03 01 00 01`	**`e` = 65537**	masked, but known from `encrypt.py`
273–276	`02 82 01 00`	INTEGER, 256 bytes	masked
277–532		`d`	**fully masked**
533–535	`02 81 81`	INTEGER, 129 bytes	masked
536–664		**`p`** (leading `00`)	**only the last 2 bytes known: `da 6b`**
665–667	`02 81 81`	INTEGER, 129 bytes	known
668–796		**`q`** (leading `00`)	**top 73 bytes known (offsets 669–741)**
797–1190		`dP`, `dQ`, `qInv`	**fully masked**

The little visible island `2msCgYEAwLGx` decodes to `da 6b 02 81 81 00 c0 b1 b1` — i.e. the tail of `p`, the DER header of `q`, and the first bytes of `q`. That is the crack the whole solve goes through.

So the usable material is:

- **`n`**: 2048 bits, of which the top 2040 are known — the low byte is masked.
- **`q`**: the **top 584 bits** of a 1024-bit prime (73 bytes × 8), leaving 440 unknown low bits.
- **`p`**: the **low 16 bits**, `0xda6b` — a free correctness check.
- `d`, `dP`, `dQ`, `qInv`: nothing.

5. Why 584 known bits of `q` is game over

This is the classic **"factoring with high bits known"** setting solved by Coppersmith's method. Writing

$$q = A + x, \quad A = q_{hi} \cdot 2^{440}, \quad 0 \le x < 2^{440}$$

we need the small root x of the monic linear polynomial

$$f(x) = x + A \pmod{n}$$

where "small root" means we recover any root bounded by roughly $N^{(\beta^2)} = N^{(1/4)}$ when the divisor we are chasing has size N^{β}, here $\beta = 1/2$.

$$N^{(1/4)} = 2^{512} > 2^{440} = \text{our unknown}$$

So knowing **more than half** the bits of one prime (512 of 1024) suffices, and 584 known bits gives 72 bits of comfortable margin. The intuition behind the challenge flavour text is exactly this: *a seal doesn't have to be whole to still remember the door it once opened.*

The complication: n is not fully known either

Coppersmith works **modulo** n, so it needs n exactly — and offset 267, the low byte of n, is masked. Since $n = p \cdot q$ with both primes odd, n is odd, which leaves only **128 candidates**. Each candidate is cheap to test: run the lattice step and check whether the returned root produces a genuine divisor of the candidate n. Only the true modulus can factor, so the sweep is self-validating.

6. Implementation

No Sage was available in the environment, so Coppersmith is implemented directly on top of `fpylll`'s LLL using the standard Howgrave-Graham construction. For a monic degree-1 f and parameters m, t the lattice is spanned by

$$g_i(x) = n^{(m-i)} \cdot f(x)^i \ (0 \le i \le m), \quad h_j(x) = x^j \cdot f(x)^m \ (1 \le j \le t)$$

with each column scaled by X^k so that short vectors correspond to polynomials that are small on $|x| \le X$.

Choosing m and t. With $X = n^{\gamma}$, $\gamma = 440/2048 \approx 0.2148$, the reduction succeeds when

$$[\, m(m+1)/2 + \gamma \cdot (\, m(m+1)/2 + t \cdot m + t(t+1)/2 \,) \,] \,/\, (m + t + 1) \;<\; \beta m = 0.5m$$

Evaluating small values: `m=4, t=4` gives `1.970 < 2.0` (tight), while **`m=5, t=5`** gives `2.437 < 2.5` — roughly 129 bits of slack in a dimension-11 lattice. That is the parameter set used.

Root finding. After LLL, row 0 encodes a polynomial `h` with `h(x₀) = 0` over the integers. Rather than fighting 5000-bit coefficients numerically, substitute `x = X·y`: the coefficients `a_k · X^k` are then all of comparable magnitude, so the roots in `y ∈ [0,1]` are numerically well behaved and `mpmath.polyroots` finds them at modest precision. Each candidate is rounded to an integer and verified **exactly** with integer arithmetic, so numerical error can never produce a false positive.

`cop.py` — the Coppersmith routine:

```
from fpylll import IntegerMatrix, LLL
from mpmath import mp, mpf, polyroots

def _build(n, A, X, m, t):
    """Howgrave-Graham lattice for f(x) = x + A mod n, divisor >= n^0.5."""
    fpow = [[1]]
    for _ in range(m):                      # f(x)^i coefficient lists
        prev = fpow[-1]
        cur = [0] * (len(prev) + 1)
        for j, c in enumerate(prev):
            cur[j]     += c * A
            cur[j + 1] += c
        fpow.append(cur)

    polys  = [[c * pow(n, m - i) for c in fpow[i]] for i in range(m + 1)]
    polys += [[0] * j + fpow[m] for j in range(1, t + 1)]

    d  = m + t + 1
    B  = IntegerMatrix(d, d)
    Xp = [X**k for k in range(d)]
    for r, cf in enumerate(polys):
        for k, c in enumerate(cf):
            B[r, k] = c * Xp[k]
    return B, d, Xp

def small_roots(n, A, X, m=5, t=5):
    B, d, Xp = _build(n, A, X, m, t)
    LLL.reduction(B)

    out = set()
    for row in range(min(3, d)):
        a = [B[row, k] // Xp[k] for k in range(d)]
        while a and a[-1] == 0:
            a.pop()
        if len(a) < 2:
            continue

        # substitute x = X*y so the coefficients are balanced
        cy = [a[k] * Xp[k] for k in range(len(a))][::-1]
        mp.dps = 200
        scale = max(abs(c) for c in cy)
        try:
            rts = polyroots([mpf(c) / mpf(scale) for c in cy],
                            maxsteps=500, extraprec=400)
        except Exception:
            continue

        for r in rts:
            if abs(mp.im(r)) > mpf(10)**-20:
                continue
            x0 = int(mp.nint(mp.re(r) * mpf(X)))
            for cand in (x0 - 1, x0, x0 + 1):          # exact integer check
                if 0 <= cand <= X and sum(c * cand**k for k, c in enumerate(a)) == 0:
                    out.add(cand)
    return sorted(out)
```

`solve.py` — the sweep over the masked byte of `n`:

```
import sys
from cop import small_roots
from Crypto.Util.number import long_to_bytes

n_hi, q_hi, p_low = [int(x) for x in open('parts.txt').read().split()]
UNK = 1024 - 584          # unknown low bits of q
A   = q_hi << UNK         # known high part of q
X   = 1 << UNK
c   = int.from_bytes(open('flag.enc', 'rb').read(), 'big')
e   = 0x10001

for b in range(1, 256, 2):            # n = p*q is odd
    n = (n_hi << 8) | b
    for x in small_roots(n, A, X, m=5, t=5):
        q = A + x
        if q > 1 and n % q == 0:
            p = n // q
            assert p & 0xffff == p_low        # matches the surviving p fragment
            d = pow(e, -1, (p - 1) * (q - 1))
            print("[+] n's masked byte = 0x%02x" % b)
            print("[+] p =", p)
            print("[+] q =", q)
            print("[+] FLAG:", long_to_bytes(pow(c, d, n)))
            sys.exit(0)
print("no root found")
```

7. Result

The sweep finishes in **5.7 seconds**:

```
[+] n's masked byte = 0x75
[+] p low 16 bits check: 0xda6b expected 0xda6b
[+] FLAG: b'HTB{r3c0v3r1ng_RSA_k3ys___l1k3___Me0w___me0o00o0o0w___Me0w}'
```

`p`:

```
131245497686548195467002565571865142560301544570020751232142961289012921376593159901052103730440250925007836457109179886547836486011666467584064970241380093385446276958000532368427648668071765509981171770398020329521298809677306252117144794197756373244812597628909948298104892770922899165466399775874596133483
```

`q`:

```
135314408378842790751605878050931209066067635249717498350882274491410611188834198512433542860977980915267615830180088553515126808474341043736459805810824761780913391116795622398843468715298669440308270490800437972694168370812052926427757058006436117836843232703533488083597801200757836873180405061962240493471
```

Note the two independent confirmations that fall out for free: the masked byte of `n` resolves to a single value `0x75`, and the recovered `p` ends in `0xda6b`, exactly matching the two-byte `p` fragment that was never used as

an input to the attack.

8. Verification

Reconstructing the full private key from `(p, q, e)` and re-exporting it reproduces the original DER byte-for-byte wherever the original survived:

```
import base64
from Crypto.PublicKey import RSA

n = p * q
d = pow(e, -1, (p - 1) * (q - 1))
pem = RSA.construct((n, e, d, p, q)).export_key().decode()
der = base64.b64decode(''.join(pem.split('\n')[1:-1]))

assert len(der) == 1191
assert all(der[i] == v for i, v in known.items())   # all 346 surviving bytes
```

```
rebuilt DER length: 1191 (expected 1191)
surviving bytes compared: 346   mismatches: 0
ciphertext < n: True
```

The seal is whole again.

9. Takeaways

- **Masked ≠ lost.** Because the redaction preserved character positions, Base64 offsets stayed intact and the rigid PKCS#1 DER layout told us precisely which field each surviving byte belonged to. Structure-aware parsing is the whole first half of this challenge.
- **Decode Base64 per-character, not per-group.** A group with 3 of 4 characters intact still yields 2 bytes. Small win here, but the habit matters when fragments are sparser.
- **Half the bits of one prime is enough.** Coppersmith's method factors a 2048-bit modulus given ~512 known bits of a 1024-bit factor. Redacting "most" of a private key is not protection; an RSA private key must be treated as all-or-nothing.
- **An unknown modulus byte is not a wall.** When a parameter the attack structurally requires is only *slightly* unknown, fold it into a brute-force loop around the attack. Here 128 candidates cost 5.7 seconds total, and the attack itself acts as the oracle for which candidate is right.
- **Never trust floating point for the final answer.** Numerics only propose candidate roots; every candidate was confirmed with exact integer arithmetic.
- And the ASCII cat hiding in the redaction was foreshadowing the flag all along: `Me0w`.

Cyber Apocalypse 2026 · Crypto · Fractured Seal — `HTB{r3c0v3r1ng_RSA_k3ys___l1k3___Me0w___me0o00o0o0w___Me0w}`