PATIENT ZERO 2

CTF Cryptography Challenge — Complete Solve Guide

RSA · Low Public Exponent · Coppersmith Small Roots · LLL Lattice Reduction

Solved by ssstrickys

FLAG RECOVERED

SDG{e93753b33118bd55f92fccab4ff912e1}

Parameter	Value
Algorithm	RSA
Exponent e	3
Modulus n	1024-bit composite
Ciphertext	1021-bit
Flag length	37 bytes
Known prefix	SDGCTF_SECURE_MSG_V1:: (22 bytes)
Known suffix	::END (5 bytes)
Attack	Coppersmith / LLL
Tool	SageMath small_roots()

1 · Challenge Overview

Patient Zero 2 presents a simulated medical-records encryption scenario. A volunteer developer encrypted patient records using RSA with the dangerously small public exponent **e = 3** and no proper padding scheme. Two files are provided:

- **encrypt.py** — the encryption script showing the exact message structure and RSA parameters
- **public.txt** — the public key (n, e), the ciphertext c, and the plaintext flag length

Provided Files

encrypt.py reveals the full encryption logic:

```
prefix = b'SDGCTF_SECURE_MSG_V1::' # 22 bytes — always known suffix = b'::END' # 5 bytes — always known padded = prefix + flag + suffix # 64 bytes total m = bytes_to_long(padded) c = pow(m, e, n) # c = m^3 mod n
```

public.txt provides n (1024-bit), e = 3, the ciphertext c (1021-bit), and flag_length = 37. Because the prefix and suffix are fixed and public, the attacker knows 27 of the 64 message bytes. Only the 37-byte flag is secret.

2 · Vulnerability Analysis

2.1 Why e = 3 is Dangerous

RSA with a small public exponent like e = 3 is fundamentally insecure when combined with unpadded or predictably-structured messages. Three classic attacks target small e:

- **Direct cube root attack** — if m is so small that $m^3 < n$, then $c = m^3$ exactly as an integer and cbrt(c) recovers m with no modular arithmetic needed
- **Broadcast attack** — if the same message is encrypted under e different public keys, the Chinese Remainder Theorem recovers m^3 over the integers
- **Coppersmith / partial plaintext attack** — if significant portions of m are known, the secret portion can be recovered via lattice reduction even with a single (n, c) pair

This challenge is the third case. The message is 512 bits (well above n^(1/3) ≈ 2^341), so the direct cube root fails. But 27 of 64 bytes are known, leaving only a 37-byte (296-bit) unknown — small enough for Coppersmith.

2.2 Message Structure as a Linear Polynomial

Because the plaintext is prefix ■ flag ■ suffix, we can express m as a linear function of the unknown flag integer x:

```
m = prefix_int · 2^((37+5)·8) + x · 2^(5·8) + suffix_int = A + B · x where: A =
bytes_to_long(prefix) · 2^336 + bytes_to_long(suffix) [constant, fully known] B = 2^40
[constant, fully known] x = bytes_to_long(flag) [unknown, 0 ≤ x < 2^296]
```

Substituting into the RSA equation $c \equiv m^3 \pmod n$ gives a degree-3 polynomial in x with a single unknown root x■:

```
f(x) = (A + B·x)³ − c ≡ 0 (mod n) Expanded: B³·x³ + 3AB²·x² + 3A²B·x + (A³ − c) ≡ 0 (mod n)
```

2.3 Coppersmith's Theorem — The Core Insight

Coppersmith (1996) proved that for any monic degree-d polynomial f over Z/nZ, a root x■ satisfying |x■| < n^(1/d) can be recovered in polynomial time via lattice reduction (LLL algorithm). The key condition:

```
|x■| < n^(1/d) For d = 3: x■ < n^(1/3) ≈ 2^341 Our x■ < 2^296 296 < 341 ✓ Coppersmith applies.
```

The bound is satisfied with 45 bits to spare, making the attack highly reliable.

3 · Attack Construction

3.1 Making the Polynomial Monic

SageMath's small_roots() requires a monic polynomial (leading coefficient = 1). Our expanded f(x) has leading coefficient $B^3 = 2^{120}$ — not 1. The standard fix is a variable substitution that shifts the problem into a monic form.

Substitute w = B·x (i.e., w = 2^40 · x):

```
f(x) = (A + B·x)³ − c Let w = B·x, so x = w / B h(w) = (A + w)³ − c = w³ + 3A·w² + 3A²·w + (A³ − c) h(w) is MONIC (leading coefficient = 1) ✓ Root: w■ = B · x■ = 2^40 · flag_int Bound: w■ < 2^40 · 2^296 = 2^336 < n^(1/3) ≈ 2^341 ✓
```

3.2 Exploiting the Flag Format to Reduce the Search Space

CTF flags follow the format SDG{...}. The inner content is 32 bytes (hex string). Knowing the first 4 bytes (SDG{) and last byte (}) reduces the unknown to just 32 bytes = 256 bits. This increases the headroom from 45 bits to 85 bits, making the attack significantly faster and more reliable:

```
flag = b'SDG{' + z + b'}' flag_int = SDG_val · 2^(33·8) + z_int · 2^8 + ord('}') Substituting into m = A + B · flag_int: m = A + B · (SDG_val · 2^264 + z_int · 2^8 + 0x7d) = (A + B·SDG_val·2^264 + B·0x7d) + B·2^8 · z_int = A2 + B2 · z_int where: A2 = A + B·(SDG_val·2^264 + 0x7d) [new known constant] B2 = B · 2^8 = 2^48 [new shift] z = 32-byte inner hex content [unknown, z_int < 2^256]
```

The monic polynomial then becomes:

```
h(w) = (A2 + w)³ − c ≡ 0 (mod n) Root: w■ = B2 · z_int < 2^48 · 2^256 = 2^304 Headroom: n^(1/3) ≈ 2^341 vs bound 2^304 => 37 bits of headroom Epsilon: epsilon = 0.04 is sufficient (lattice stays small, runs fast)
```

3.3 How Coppersmith / LLL Works Internally

The Howgrave-Graham (1997) reformulation of Coppersmith's method works as follows:

- **Lattice construction** — Build a set of auxiliary polynomials g_{i,j}(w) = n^(m–i) · w^j · h(w)^i. These all share the root w■ modulo a power of n
- **Scale columns** — Multiply column k by X^k (where X = 2^304) so that small coefficients correspond to small roots
- **LLL reduction** — Run the Lenstra-Lenstra-Lovász lattice basis reduction algorithm. LLL finds a short vector in the lattice, which corresponds to a polynomial with small coefficients
- **Howgrave-Graham bound** — If the short vector's Euclidean norm is less than n^m / sqrt(dim), then the corresponding polynomial has w■ as an integer root (not just modular)
- **Root extraction** — Solve the resulting small polynomial over Z to get w■ exactly, then recover z = w■ / B2 and reconstruct the flag

The epsilon parameter controls the trade-off: smaller epsilon builds a larger lattice (stronger but slower), larger epsilon builds a smaller lattice (faster but requires more headroom). With 37 bits of headroom, epsilon = 0.04 gives a lattice of manageable dimension that completes in seconds.

4 · Complete Solve Script (SageMath)

The final solve script is 25 lines of pure SageMath with no external dependencies. Run it at sagecell.sagemath.org or any local Sage installation:

```
n = 10806003193126635375880133078247363932003922520131191717844970501917666069624487235127138248686450737760780753861806284766511556202
c = 15326938655178168652190890963913402575115995599841955649191058277858316963717284065315665307742177774724795623631408616667745883865
```

```
prefix    = b'SDGCTF_SECURE_MSG_V1::'
suffix    = b'::END'
flag_len  = 37

# Build A and B for m = A + B*flag_int
def b2i(b): return Integer(int.from_bytes(b, 'big'))
B  = Integer(2)^(len(suffix)*8)                                    # 2^40
A  = b2i(prefix) * Integer(2)^((flag_len+len(suffix))*8) + b2i(suffix)

# Exploit SDG{} format: flag = SDG{ + z(32 bytes) + }
# New constants so unknown z_int < 2^256
SDG = b2i(b'SDG{')
B2  = B * Integer(2)^8                                             # 2^48
A2  = A + B * (SDG * Integer(2)^(33*8) + ord('}'))

# Monic polynomial h(w) = (A2 + w)^3 - c, root w0 = B2*z < 2^304
PR.<w> = PolynomialRing(Zmod(n))
h = (A2 + w)^3 - c

# Coppersmith small roots — runs in ~10 seconds
roots = h.small_roots(X=2^304, beta=1, epsilon=0.04)

for r in roots:
    if int(r) % int(B2) == 0:
        z      = int(r) // int(B2)
        z_bytes = z.to_bytes(32, 'big')
        flag   = b'SDG{' + z_bytes + b'}'
        m      = int.from_bytes(prefix + flag + suffix, 'big')
        if pow(m, 3, n) == c:
            print('FLAG:', flag.decode())
```

Expected Output

```
Monic: True Running small_roots... Roots:
[12887236520147591464921203182986708751941861084017317491466798187598435571572425429941747712] FLAG:
SDG{e93753b33118bd55f92fccab4ff912e1}
```

5 · Step-by-Step Walkthrough

Step 1: Reconnaissance

- Unzip the archive to get encrypt.py and public.txt.
- Read encrypt.py: RSA with e=3, message = prefix + flag + suffix, no padding.
- Read public.txt: n (1024-bit), e=3, c (1021-bit), flag_length=37.
- Identify the attack surface: small exponent + known plaintext structure.

Step 2: Classify the Attack

- Message is 512 bits. n^(1/3) ≈ 2^341. Since 512 > 341, direct cube root fails — m^3 wraps around n.
- Only one ciphertext exists, so broadcast attack is impossible.
- 27 of 64 bytes are known (prefix + suffix). Coppersmith partial-plaintext attack applies.
- Verify: unknown flag is 37 bytes = 296 bits < 341 = n^(1/3). The bound is met.

Step 3: Construct the Polynomial

- Express m = A + B*x where A encodes prefix and suffix (known) and x = flag_int (unknown).
- A = prefix_int * 2^(42*8) + suffix_int = fixed 511-bit constant.
- B = 2^40 (shift to make room for suffix bytes).
- Substituting: c ≡ (A + B*x)^3 (mod n) => f(x) = (A+B*x)^3 - c ≡ 0 (mod n).

Step 4: Monicize via Substitution

- f(x) has leading coefficient B^3 = 2^120. SageMath small_roots() requires monic.
- Substitute w = B*x. Then h(w) = (A+w)^3 - c is monic (leading coeff = 1).
- New root: w0 = B * x0 < 2^40 * 2^296 = 2^336 < n^(1/3) ≈ 2^341. Still valid.

Step 5: Reduce Search Space Using SDG{} Format

- CTF flag format is SDG{<32 hex chars>}. First 4 and last 1 bytes are known.
- Rewrite flag_int = SDG_val * 2^264 + z_int * 2^8 + 0x7d where z_int < 2^256.
- Fold known parts into a new constant A2. Unknown reduces to z_int < 2^256.
- New shift B2 = 2^48. New root bound: w0 = B2*z < 2^304. Headroom = 37 bits.
- This allows epsilon=0.04 — a small, fast lattice that finishes in ~10 seconds.

Step 6: Run Coppersmith in SageMath

- Build h(w) = (A2+w)^3 - c over Zmod(n).
- Call h.small_roots(X=2^304, beta=1, epsilon=0.04).
- SageMath internally builds the LLL lattice, reduces it, and extracts the small root w0.
- w0 = 1288723652014759146492120318298670875194186108401731749146679818759843557157242542994174771 2

Step 7: Recover the Flag

- Check w0 % B2 == 0 (it is — since w0 = B2 * z0 exactly).
- Compute z0 = w0 // B2.
- Convert: z_bytes = z0.to_bytes(32, 'big').
- Reconstruct: flag = b'SDG{' + z_bytes + b'}'.
- Verify: pow(bytes_to_long(prefix+flag+suffix), 3, n) == c. Confirmed.
- FLAG: SDG{e93753b33118bd55f92fccab4ff912e1}

6 · Debugging Journey & Why Naive Approaches Failed

The path to the solution involved several failed approaches. Understanding why they failed is as instructive as the solution itself.

✗ Direct Integer Cube Root [FAILED]

What happened: Tried cbrt(c) directly. c is 1021 bits; its cube root is ~341 bits. But m is 512 bits. Since 512 ≠ 341, this fails immediately — m^3 is reduced mod n so c ≠ m^3 as integers.

Why: c = m^3 mod n, not m^3 over Z. Modular reduction destroys the integer cube root shortcut.

✗ Iterating k in c + k*n [FAILED]

What happened: Tried cube root of c + k*n for k = 0, 1, 2, ... hoping one would be an exact cube. This would work if m^3 = c + k*n for small k, but k can be up to ~2^512. Not feasible.

Why: k is astronomically large — up to 2^512. No shortcut exists without knowing the structure.

✗ Naive Coppersmith with fpylll (Python) [FAILED]

What happened: Attempted to implement Coppersmith manually using fpylll's LLL with X=2^336. Matrix entries reached 2^2958 bits — far beyond floating-point precision. fpylll silently produced wrong results. Newton's method on the reduced poly converged to spurious roots every time.

Why: fpylll uses floating-point GSO internally. For entries >~2^500, precision loss causes silent failure.

✗ SageMath small_roots with epsilon=0.05 (first attempt) [FAILED]

What happened: Polynomial h(w) = (A+w)^3 - c with X=2^336 and epsilon=0.05. Sage returned []. Tried epsilon down to 1/50. All returned [].

Why: With only 5 bits of headroom (2^336 vs n^(1/3)~2^341), epsilon must be tiny, requiring an enormous lattice. SageMathCell times out silently and returns [].

✓ SageMath small_roots — SOLVED with SDG{} reduction [SUCCESS]

What happened: Exploited the known SDG{...} flag format to reduce the unknown to 256 bits. New bound 2^304 gives 37 bits of headroom. epsilon=0.04 works with a small lattice. Sage found the root in ~10 seconds.

Why: The extra known bytes from SDG{} format reduced the search space by 2^40, providing enough headroom for a practical epsilon value.

7 · Key Concepts Reference

7.1 RSA Fundamentals

```
Key generation: n = p·q, ϕ(n) = (p−1)(q−1) e chosen s.t. gcd(e, ϕ(n)) = 1 d = e^(−1) mod ϕ(n) Encryption: c = m^e mod n Decryption: m = c^d mod n Security relies on: difficulty of factoring n AND e being large/random
```

7.2 Why Small e is Dangerous

With e=3 and no padding, an attacker who knows part of the plaintext can exploit the algebraic structure of RSA. The encryption function f(x) = (A + B·x)^3 mod n is a low-degree polynomial in the unknown x. Coppersmith showed that such polynomials have their small roots efficiently recoverable via lattice methods.

7.3 LLL Algorithm Summary

The Lenstra-Lenstra-Lovász (LLL) algorithm takes a lattice basis and returns a reduced basis where the first vector is provably short (within a 2^(n/2) factor of the shortest lattice vector). For Coppersmith's method, LLL finds a polynomial with small coefficients that shares the secret root — converting a modular problem into an integer problem solvable by standard root-finding.

7.4 The Howgrave-Graham Condition

```
Given h(x) with root x■ mod n, and ||h(x·X)|| < n^m / sqrt(dim) then h(x■) = 0 holds over Z (not just mod n) This converts the modular small-root problem into an integer polynomial root-finding problem, which is trivially solved.
```

7.5 Coppersmith Parameter Guide

epsilon	Lattice dim	Headroom needed	Speed
0.10	~10	>100 bits	Instant
0.05	~40	>50 bits	Seconds
0.04	~83	>37 bits	~10s
0.02	~300	>15 bits	Minutes
0.01	~1200	>5 bits	Hours/timeout

8 · How to Fix This Vulnerability

The vulnerable developer made several mistakes that compound each other. Any one of the following fixes would have prevented this attack:

✓ Use Proper RSA Padding (OAEP)

OAEP (Optimal Asymmetric Encryption Padding) randomises the message before encryption. Even with e=3 and known prefix/suffix, the attacker sees a different ciphertext every time and the algebraic structure is destroyed. This is the correct fix.

```
from Crypto.Cipher import PKCS1_OAEP; cipher = PKCS1_OAEP.new(key); c = cipher.encrypt(flag)
```

✓ Use a Larger Exponent

e = 65537 (0x10001) is the standard choice. With e = 65537, even a fully known plaintext structure provides no algebraic shortcut — Coppersmith's bound n^(1/e) = n^(1/65537) is astronomically small.

```
e = 65537 # standard safe choice
```

✓ Do Not Use RSA for Symmetric Data

RSA is an asymmetric primitive designed to encrypt small keys, not bulk data. The correct pattern is hybrid encryption: generate a random AES key, encrypt the data with AES-GCM, and encrypt only the AES key with RSA-OAEP.

```
# AES-GCM for data + RSA-OAEP for key transport
```

✓ Randomise the Message

Prepend a random nonce to the message before encryption so that identical plaintexts produce different ciphertexts. This defeats broadcast and algebraic attacks even without proper padding.

```
nonce = os.urandom(16); padded = nonce + prefix + flag + suffix
```

9 · Summary

Patient Zero 2 demonstrates a real-world class of cryptographic vulnerability: **RSA with a small public exponent and structured plaintext**. The attack chain is:

- Recognise that e=3 + known prefix/suffix creates a low-degree polynomial problem
- Construct f(x) = (A + B·x)^3 − c ≡ 0 (mod n) where x is the unknown flag
- Apply Coppersmith's theorem: since |x■| < n^(1/3), the root is recoverable by LLL
- Make the polynomial monic via the substitution w = B·x
- Exploit the SDG{} flag format to shrink the unknown from 296 to 256 bits, enabling epsilon=0.04
- Run SageMath small_roots() — finds the root in ~10 seconds
- Recover the flag: SDG{e93753b33118bd55f92fccab4ff912e1}

FINAL FLAG

SDG{e93753b33118bd55f92fccab4ff912e1}