repeated-rsa

We were given three RSA moduli n1, n2, n3, all with the same public exponent e=65537/

Decrypt in reverse order of encryption: first invert the last encryption (mod n3), then the second (mod n2), then the first (mod n1). The final value is the plaintext; convert to bytes.

step 1: Compute gcd(n1, n2), gcd(n1, n3), gcd(n2, n3). if any of these is greater than 1 then the two moduli share a prime factor. in this challenge they do, so factoring becomes trivial.
step 2: Once we find a common prime p, we divide the modulus by p to get the other prime. that way we fully factor each modulus.
step 3: For each modulus we compute phi = (p-1)(q-1) and then the private exponent d = e inverse mod phi.
step 4: Since the message was encrypted in the order n1 → n2 → n3, we decrypt in reverse. first undo the n3 encryption using d3, then undo n2 with d2, finally undo n1 with d1.
step 5: The result is the plaintext message as an integer. we convert it to bytes and decode it as ascii to get the flag.

solver.py

```
#!/usr/bin/env python3
from math import gcd
from Crypto.Util.number import inverse, long_to_bytes

# Given challenge values (unchanged)
c =
22294107855279463176353647748352925374834304539844959685744948314890475115195831264388104064157805785893177280308760719601282029636999116410719094831035646341786103539063832685153414254210352637538317254003034392774677173879944716708844345806661685162757591152485908334117215483454304081970498934519824604061749779970095035268226700546848779926276812909412304033737629551192402448454462953942161502288872062996569920734979494404696622257100766685119573316535902606153701523873795097254686504731207424799089794756788426424905628957464816167161517091505355537084172446679447485116547136235049479797819458319251879738 1

n1 =
13852404208507408802969880487839186284776154658591414025042810686360400013127774847502340273305623845274604456776137166609277405684272007689391356639812356940155331180589738056370147100521557117036516708000178035092048663373455750305534323237059838602263431659551056045872580098364916417474039437912346894641330241402004752256659954856461727024372071766854017300857517546611231762952091418981114953473216240892822633582203036320527992573432136583591014349994351427032781965145608630820018051050230075720564731871368372231470262203086280844504502775043017498384280801239614982205079609753900100398376733169246
```

```
631316697
n2 =
157481526377963653681377977065279962986082153378904569272690384279268466746918759437202672449713703811418099684073262758718033238035817470803142615443324859437313688526550678508964715166250291431121656278120774730646084407106601752367592482409483408141850514487036476320582861163973976910707557410605834905206690702446884352375408555524956535398784520181674711320516932786596840516537817097470037648423645591703992855238721222662257745670966410205056143770381670722576280926019651419684086853909406555382387778305262546640095920491493653227016020306889988033735787510505133950884653729358378413986957095383268339886631316697
n2 =
157481526377963653681377977065279962986082153378904569272690384279268466746918759437202672449713703811418099684073262758718033238035817470803142615443324859437313688526550678508964715166250291431121656278120774730646084407106601752367592482409483408141850514487036476320582861163973976910707557410605834905206690702446884352375408555524956535398784520181674711320516932786596840516537817097470037648423645591703992855238721222662257745670966410205056143770381670722576280926019651419684086853909406555382387778305262546640095920491493653227016020306889988033735787510505133950884653729358378413986957095383268339886631316697
n3 =
103508961771756820075115988050288476001379574871173020627748131387633518735934619804485072729759989224683047984922313432077617210527079257866057822623844437684432191232625777135115948395948668451848278421528080317284893810105074403857620731191856291489784467171676302784239618141391876550477219253340936480396121449782952047964419764656692261340914971802008230864767428777901993625297384463787428487797182685041626703335727174060017929541747701415809353964695989591957864738855846553344511059103429847080406008743915928231239128198427357731719909272747212452582738103464050966132078976537967870229167575972255226511119
e = 65537

def find_common_factors(mods):
    """Return a dict mapping discovered prime -> list of indices of moduli
that contain it."""
    n = len(mods)
    common = {}
    for i in range(n):
        for j in range(i+1, n):
            g = gcd(mods[i], mods[j])
            if g != 1:
                common.setdefault(g, []).extend([i, j])
    # normalize lists to unique indices
    for k in list(common.keys()):
        common[k] = sorted(set(common[k]))
    return common

def factor_moduli(mods, common_factors):
    """Return a list of (p, q) for each modulus index i."""
    n = len(mods)
    factors = [None]*n
    # For each discovered common prime, divide it out of moduli that contain
it
    for p, indices in common_factors.items():
        for i in indices:
            if factors[i] is None:
                if mods[i] % p != 0:
                    raise ValueError("Unexpected: modulus not divisible by
found gcd")
```

```
                q = mods[i] // p
                factors[i] = (p, q)
    # If any modulus remains unfactored, try to factor it by using any known
primes (rare here)
    for i in range(n):
        if factors[i] is None:
            # try dividing by any found primes
            for p in common_factors:
                if mods[i] % p == 0:
                    factors[i] = (p, mods[i] // p)
                    break
    # Final sanity check
    if any(x is None for x in factors):
        raise RuntimeError("Could not factor all moduli with discovered
gcds.")
    return factors

def compute_private_exponent(p, q, e):
    phi = (p-1)*(q-1)
    return inverse(e, phi)

# main
mods = [n1, n2, n3]
common = find_common_factors(mods)
if not common:
    raise RuntimeError("No common prime factors found between moduli;
different attack required.")

pairs = factor_moduli(mods, common)

# Compute private exponents in the same index order (matching n1,n2,n3)
ds = [compute_private_exponent(p, q, e) for (p,q) in pairs]

# Decrypt in reverse order of encryption: last applied modulus is n3, so
invert with d3 mod n3 first
step_after_n3 = pow(c, ds[2], n3)
step_after_n2 = pow(step_after_n3, ds[1], n2)
m_int = pow(step_after_n2, ds[0], n1)

flag = long_to_bytes(m_int)
print(flag.decode())
```