RSA? (Crypto, 2p)

###ENG PL

We get a ciphertext that seems to be encrypted via RSA from openssl commandline. We also have access to the public key, and therefore we proceed like with standard RSA cipher, by recovering parameters:

```
e = 3
```

```
n = 23292710978670380403641273270002884747060006568046290011918413375473934024039715180540887338067
```

And using YAFU we factor the modulus into:

```
p = 26440615366395242196516853423447
```

```
q = 27038194053540661979045656526063
```

```
r = 32581479300404876772405716877547
```

We get 3 prime numbers. This is still fine, this could simply be multiprime RSA. There is nothing fancy about it, the totient is simply `(p-1)(q-1)(r-1)` and rest of the calculation goes as usual. But it doesn't, because we find that modular multiplicative inverse does not exist. Reason for this is apparent: `gcd(e, totient) = 3` and it should be 1. This is not the first time we encounter similar case (see https://github.com/p4-team/ctf/tree/master/2015-10-18-hitcon/crypto_314_rsabin#eng-version) so we have some idea of how to approach this.

We need to get rid of this 3 before we could apply RSA decoding. This means the encryption is:

```
ciphertext = plaintext^e mod n = (plaintext^e')^3 mod n
```

So if we could peform a modular cubic root (mod n) of both sides of the equation we could then apply RSA decoding with `e' = e/3`. Here is't even easier since `e=3` and therefore `e' = e/3 = 1` which means our encryption is simply:

```
ciphertext = plaintext^3 mod n
```

So the whole decryption requires modular cubic root (mod n) from ciphertext.

Some reading about modular roots brings us to conclusion that it's possible to do, but only in finite fields. So it can't be done for `n` that is a composite number, which we know it is since it's `p*q*r`.

This problem brings to mind Chinese Reminder Theorem (https://en.wikipedia.org/wiki/Chinese_remainder_theorem) We consider this for a while and we come up with the idea that if we could calculate cubic modular root from ciphertext (mod prime) for each of our 3 primes, we could then calcualte the combined root. We can to this with Gauss Algorithm (http://www.di-mgt.com.au/crt.html#gaussalg)

So we proceed and calculate:

```
pt^3 mod p = ciperhtext mod p = 20827907988103030784078915883129
```

```
pt^3 mod q = ciperhtext mod q = 19342563376936634263836075415482
```

```
pt^3 mod r = ciperhtext mod r = 10525283947807760227880406671000
```

And then it took us a while to come up with solving this equations for pt (publications mention only some special cases for those roots...) Finally we figured that wolframalpha had this implemented, eg:

http://www.wolframalpha.com/input/?i=x^3+%3D+20827907988103030784078915883129+%28mod+26440615366395242196516853423447%29

This gives us a set of possible solutions:

```
roots0 = [5686385026105901867473638678946, 7379361747422713811654086477766, 13374868592866626517389128266735]
roots1 = [19616973567618515464515107624812]
roots2 = [6149264605288583791069539134541, 13028011585706956936052628027629, 13404203109409336045283549715377]
```

We apply Gauss Algoritm to those roots:

```
def extended_gcd(aa, bb):
    lastremainder, remainder = abs(aa), abs(bb)
    x, lastx, y, lasty = 0, 1, 1, 0
    while remainder:
        lastremainder, (quotient, remainder) = remainder, divmod(lastremainder, remainder)
        x, lastx = lastx - quotient * x, x
        y, lasty = lasty - quotient * y, y
    return lastremainder, lastx * (-1 if aa < 0 else 1), lasty * (-1 if bb < 0 else 1)

def modinv(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise ValueError
    return x % m

def gauss(c0, c1, c2, n0, n1, n2):
    N = n0 * n1 * n2
    N0 = N / n0
    N1 = N / n1
    N2 = N / n2
    d0 = modinv(N0, n0)
    d1 = modinv(N1, n1)
    d2 = modinv(N2, n2)
    return (c0*N0*d0 + c1*N1*d1 + c2*N2*d2) % N

roots0 = [5686385026105901867473638678946, 7379361747422713811654086477766, 13374868592866626517389128266735]
roots1 = [19616973567618515464515107624812]
roots2 = [6149264605288583791069539134541, 13028011585706956936052628027629, 13404203109409336045283549715377]


for r0 in roots0:
    for r1 in roots1:
        for r2 in roots2:
            M = gauss(r0, r1, r2, p, q, r)
            print long_to_bytes(M)
```

Which gives us the flag for one of the combinations: `0ctf{HahA!Thi5_1s_n0T_rSa~}`

###PL version

Dostajemy zaszyfrowany tekst oraz informacje która wskazywałaby że szyfrowano go za pomocą RSA przy pomocy openssl. Dostajemy także klucz publiczny, więc postępujemy tak jak w klasycznym RSA, zaczynając od odzyskania parametrów klucza:

```
e = 3
```

```
n = 23292710978670380403641273270002884747060006568046290011918413375473934024039715180540887338067
```

A za pomocą YAFU dokonujemy faktoryzacji modulusa:

```
p = 26440615366395242196516853423447
```

```
q = 27038194053540661979045656526063
```

```
r = 32581479300404876772405716877547
```