RSAnt - Writeup

Author: Luca Boscarato

Challenge Source Code:

```
from random import *
from Crypto.Util.number import *
from math import gcd
flag = bytes_to_long(b"RICK{NeverGonnaGiveYouUp}")

def encrypt():
    tmp = randint(2**1023, 2**1024)
    p = next_prime(1337*tmp + randint(2, 2**512))
    q = next_prime(7331*tmp + randint(2, 2**512))
    N = p*q
    return N

def l3ak(n):
    print('Security Alert!!')
    print('There is a L3AKER l3aking our data!! [~] :/\n')
    c1 = pow(bytes_to_long(b"factoring modulus?"), e, n)
    c2 = pow(bytes_to_long(b"without the modulus?"), e, n)
    return c1, c2

e = 65537
n = encrypt()
enc = pow(flag, e, n)
c1, c2 = l3ak(n)

print(f'Encrypted flag = {enc}\n')
print(f'c1 = {c1}\n')
print(f'c2 = {c2}\n')
```

Information Retrieval

This challenge has to do with RSA. However, this is not a classical RSA challenge, since the value we are provided with are the encrypted flag and two ciphertexts of which the plaintext is known. Therefore, to make a recap, let us list all the elements we have at disposal:

- Public exponent e - We know that the value for e is 65537
- We have two couples (plaintext,ciphertext)
- We have the encrypted flag
- How the modulo n was generated

Exploitation

At a first glance, the name of the function *l3ak* is suspicious - let us start from it. The function encrypts two messages and returns the correspondant ciphertexts. What one should notice here is that for both operations, both the ciphertext and plaintext value is known. To get the plaintext value we just reproduce the same computation done by the server:

```
m1 = bytes_to_long(b"factoring modulus?") #8918592752769306591549842352178849425748799
m2 = bytes_to_long(b"without the modulus?") #681721620536571024375232508196690023732368208703
```

Can we perhaps carry out a known plaintext attack in this case? The answer to the question is yes. In particular, since the public exponent used is not extremely large (65537), it is sufficient to compute `gcd(m1**e−c1,m2**e−c2)` and the result will be $k * n,\ k \in Z$. Therefore, as a result of the computation of the *gcd*, we will get a multiple of *n* (notice that *k* can also be 1 in some cases). Furthermore, notice that *k* is extremely likely to be small, therefore we can simply try small values up until we get a correspondance for: `pow(m1,e,n) == c1`.

Perfect, now we have successfully retrieved the modulo *n*, what is the next step? In order to decrypt the flag, we necessarily need the private exponent *d*, which can not be retrieved by any kind of attack given the provided couple (*e*,*n*). What we can do, though, is try to find a way to factor *n*. In particular, if one looks at the encrypt function, it is clear that *p* is a prime number very close to `1337*tmp` and q is a prime number very close to `7331*tmp` (since in both cases we are adding at most 2^{512}, which is insignificant compared to 2^{1023} or 2^{1024}). From this, we deduce:

```
tmp = isqrt((n)/(1337*7331))
```

According to what we stated before, we can approximate our *q* like this:

```
q_approx = 7331*tmp - 2**513
```

N.B. we remove 2513 to avoid overestimating the q.**
Now we need to find the “missing” part. The question we ask ourselves is: “Which is the number that added to our *q_approx* lets us get the correct value of q”? This question can be answered via Coppersmith, in this way:

```
F.<x> = PolynomialRing(Zmod(n), implementation='NTL')
f = x - q_approx
roots = f.small_roots(X=2**512, beta=0.5)
```

The general idea is that we build a polynomial `f` starting from our `q_approx` and trying to find the roots of `x-q_approx`, in which `x` has to be a 512 bits number (given the information we retrieved from the encrypt function). If such an `x` is found, we have successfully found the “missing” part of the number. Notice that the value for beta should be found via trial and error.
So we can compute our original *q*:

```
q = q_approx-delta
```

With this being done, we have basically solved the challenge because we can just apply the standard equalities for RSA and retrieve our parameters:

```
p = int(n)//int(q)
d = inverse_mod(65537, (p-1)*(q-1))
```

Lastly, we get our well deserved flag:

```
print(long_to_bytes(int(pow(ciphertext,d,n))))
```

Additional notes

During the competition, we have experienced troubles trying to compute the exponentiation of `m1^e` using the `pow` function from python. This is likely due to the internal implementation of the function and how it behaves when big integers are to be computed. To avoid this issue, one can use the default python exponentiation `m1**e`.

Full script

```
from Crypto.Util.number import *
from math import gcd


e = 65537


ciphertext = 17929968684899453914112238296223003774438449762160319812407700399163631976960356406281709235360090659120747625132728227769511026870461512006865477618350720451270276684793012590050065998305018527434489902577972652356528003290819476335694454982496321533941903854339043965521074151014300358273799661777089569216230628606442171889279721376523730092692567515307489453293593729129580215730572485632158451713122694014019049115770989442168319676670767259315440626305443690056623146471910231297324746651041877003919400087889751583898562346117678262826136577161969885929576745116250415101072805659469511704066102332796802345151980456 4
c1 = 123377882241082251325350065419048806024607608617091688983830323277604694641240653374439302807138122463323990418348747537591347656262195555533468416119550902963546537686539626424151682647446653286178409257716563958279306978514921053134482777035689209563599648603507287706662378931300585730154813000147325737518201568052630101704051194396183990174893718130243703251255733422231215114594356086085529845721945162634747194330158630722370297888040809287372214916411109961066039419282486341414973486625115394106601990911459069075609672097833893961312251252396858703096951885954537748265354572684903035976149443662164781331591137509
c2 = 216996462348882812127549126696128848915007479735759067925583940914005783704777739383583486500248026532990006813147882357469539758674416271697210195684094572116588619081813131930186778045755089586040232978039441432232050042271853194470413044599117641325812011411950889172242559099135026172425733801215110286748370156962444409678069139574351948837982999861835569715249682828276876910129256029485173183675521926552841364713489224945222100475996799601024631599522598983661501357640196487134275394097207489818217501235732511301942649024362999538093008833593968191799917057618382726046836320499881842515455356124359654138411130319


m1 = bytes_to_long(b"factoring modulus?")
m2 = bytes_to_long(b"without the modulus?")

ct_1 = m1**e
ct_2 = m2**e

n = gcd(ct_1-c1, ct_2-c2)//2

tmp = isqrt((n)/(1337*7331))
q_approx = 7331*tmp - 2**513
F.<x> = PolynomialRing(Zmod(n), implementation='NTL')
f = x - q_approx



roots = f.small_roots(X=2**512, beta=0.5)
for delta in roots:
    q = q_approx-delta
    p = int(n)//int(q)
    d = inverse_mod(65537, (p-1)*(q-1))

    print(long_to_bytes(int(pow(ciphertext,d,n))))
```