RSA (Crypto, 113p)

###ENG PL

In the task we get source code of encryption routine, a public key and encrypted flag. The source code is quite simple:

```
import gmpy
from Crypto.Util.number import *
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

flag = open('flag', 'r').read() * 30

def ext_rsa_encrypt(p, q, e, msg):
    m = bytes_to_long(msg)
    while True:
        n = p * q
        try:
            phi = (p - 1)*(q - 1)
            d = gmpy.invert(e, phi)
            pubkey = RSA.construct((long(n), long(e)))
            key = PKCS1_v1_5.new(pubkey)
            enc = key.encrypt(msg).encode('base64')
            return enc
        except:
            p = gmpy.next_prime(p**2 + q**2)
            q = gmpy.next_prime(2*p*q)
            e = gmpy.next_prime(e**2)

p = getPrime(128)
q = getPrime(128)
n = p*q
e = getPrime(64)
pubkey = RSA.construct((long(n), long(e)))
f = open('pubkey.pem', 'w')
f.write(pubkey.exportKey())
g = open('flag.enc', 'w')
g.write(ext_rsa_encrypt(p, q, e, flag))
```

Initially the algorithm generates a very small 128 bit primes and constructs modulus from them. This is the modulus value from the public key we get. However the algoritm is not using this small value for encryption since the data are too long - it uses `gmpy.next_prime()` to get bigger prime numbers. The problem is that `next_prime` is deterministic so if we can get the initial `p` and `q` values we can simply perform the same kind of iteration to get the `p` and `q` used for encrypting the flag.

Since the initial modulus is just 256 bit we can factor it with YAFU to get the initial primes. We then simply perform the same loop to get final primes, we use modinv to get decryption exponent and then simply decrypt the flag:

```
import base64
import codecs
import gmpy
import math
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

n = 980994077679753602906602271171260570145371574681916544264112304684890430099 77
p = 311155972145869391293781528370734636009 # from YAFU
q = 315274063651866931016337573625089033553
e = 12405943493775545863


def long_to_bytes(flag):
    return "".join([chr(int(flag[i:i + 2], 16)) for i in range(0, len(flag), 2)])


def bytes_to_long(str):
    return int(str.encode('hex'), 16)


with codecs.open("./flag.enc") as flag:
    data = flag.read()
    msg = base64.b64decode(data)
    print('msg', math.log(bytes_to_long(msg), 2))
    while True:
        print("looping")
        try:
            print('p*q', math.log(long(p), 2)+math.log(long(q), 2))
            phi = (p - 1) * (q - 1)
            d = gmpy.invert(e, phi)
            key = RSA.construct((long(n), long(e), long(d)))
            algo = PKCS1_v1_5.new(key)
            decrypted = algo.decrypt(msg, 0x64)
            print(decrypted)
            print(p, q, e)
            print(long_to_bytes(decrypted))
        except Exception as ex:
            print(ex)
            p = gmpy.next_prime(p ** 2 + q ** 2)
            q = gmpy.next_prime(2 * p * q)
            e = gmpy.next_prime(e ** 2)
            n = long(p)*long(q)
```

###PL version

W zadaniu dostajemy kod źródłowy szyfrowania, klucz publiczny oraz zaszyfrowaną flagę. Kod jest dość prosty:

```
import gmpy
from Crypto.Util.number import *
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

flag = open('flag', 'r').read() * 30

def ext_rsa_encrypt(p, q, e, msg):
    m = bytes_to_long(msg)
    while True:
        n = p * q
        try:
            phi = (p - 1)*(q - 1)
            d = gmpy.invert(e, phi)
            pubkey = RSA.construct((long(n), long(e)))
            key = PKCS1_v1_5.new(pubkey)
            enc = key.encrypt(msg).encode('base64')
            return enc
        except:
            p = gmpy.next_prime(p**2 + q**2)
            q = gmpy.next_prime(2*p*q)
            e = gmpy.next_prime(e**2)

p = getPrime(128)
q = getPrime(128)
n = p*q
e = getPrime(64)
```

```
pubkey = RSA.construct((long(n), long(e)))
f = open('pubkey.pem', 'w')
f.write(pubkey.exportKey())
g = open('flag.enc', 'w')
g.write(ext_rsa_encrypt(p, q, e, flag))
```

Początkowo algorytm generuje bardzo małe 128 bitowe liczby pierwsze i buduje z nich modulusa. To jest wartość modulusa w kluczu publicznym, który dostajemy. Niemniej jednak algorytm nie używa tych małych liczb do szyfrowania ponieważ liczba danych jest za duża - zamiast tego używa `gmpy.next_prime()` żeby znaleźć większe liczby pierwze. Problem z tym rozwiązaniem polega na tym, że `next_prime` jest deterministyczne więc jeśli znamy początkowe wartości `p` oraz `q` możemy wykonać taką samą iteracje i wyzaczyć finalne `p` oraz `q` użyte do szyfrowania flagi.

Ponieważ znany modulus ma tylko 256 bitów możemy go faktoryzować za pomocą YAFU aby dostać początkowe wartości liczb pierwszyh. Następnie po prostu wykonujemy identyczną pętlę jak w kodzie z zadania aby wyznaczyć finalne wartości liczb pierwszych, używamy modinv żeby wyliczyć wykładnik deszyfrujący a następnie dekodujemy flagę:

```
import base64
import codecs
import gmpy
import math
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_v1_5

n = 980994077679753602906602271171260570145371574681916544264112304684890430099 77
p = 311155972145869391293781528370734636009 # from YAFU
q = 315274063651866931016337573625089033553
e = 12405943493775545863


def long_to_bytes(flag):
    return "".join([chr(int(flag[i:i + 2], 16)) for i in range(0, len(flag), 2)])


def bytes_to_long(str):
    return int(str.encode('hex'), 16)


with codecs.open("./flag.enc") as flag:
    data = flag.read()
    msg = base64.b64decode(data)
    print('msg', math.log(bytes_to_long(msg), 2))
    while True:
        print("looping")
        try:
            print('p*q', math.log(long(p), 2)+math.log(long(q), 2))
            phi = (p - 1) * (q - 1)
            d = gmpy.invert(e, phi)
            key = RSA.construct((long(n), long(e), long(d)))
            algo = PKCS1_v1_5.new(key)
            decrypted = algo.decrypt(msg, 0x64)
            print(decrypted)
            print(p, q, e)
            print(long_to_bytes(decrypted))
        except Exception as ex:
            print(ex)
            p = gmpy.next_prime(p ** 2 + q ** 2)
            q = gmpy.next_prime(2 * p * q)
            e = gmpy.next_prime(e ** 2)
            n = long(p)*long(q)
```