NSUCRYPTO 2020: Hidden RSA

Ioan Dragomir[1], Gabriel Tulba-Lecu[2], and Mircea-Costin Preoteasa[3]

[1] ioandr@gomir.pw – Technical University of Cluj-Napoca
[2] gabi_tulba_lecu@yahoo.com – Polytechnic Univeristy of Bucharest
[3] mircea_costin84@yahoo.com – Polytechnic Univeristy of Bucharest

Table of Contents

1 Problem summary 1
2 Solution 2
2.1 Recovering n 2
2.2 Finding e 2
2.3 RSA internals 3
2.4 Factoring n 3
2.5 Decrypting the message 3
A Python code (requires yafu) 4

1 Problem summary

Bob gives us an RSA encryption oracle. He keeps all the RSA parameters secret, even e and n, which are normally public. The encryption oracle allows us to evaluate:

$$E(x) = x^e \mod n$$

for any $x \in [0, 10^{70}) \cap \mathbb{N}$. Given

$$E(m) = 715118966813248334583613928851843449333331598308638786001892120737775821781 73$$

we are supposed to find m.

2 Solution

2.1 Recovering n

We notice the magnitude of most outputs to be around 2^{255}. This indicates that n is likely 256 bits long, which is a conventional value, but very low for today's security standards. As a first step, we use the property that RSA is homomorphic under multiplication:

$$E(a) \cdot E(b) \equiv E(a \cdot b) \mod n$$

to obtain a system of equations of the form

$$\begin{cases} E(a_1) \cdot E(b_1) - E(a_1 \cdot b_1) = n \cdot k_1 \\ E(a_2) \cdot E(b_2) - E(a_2 \cdot b_2) = n \cdot k_2 \\ \dots \end{cases}$$

where all values in the left hand side are known by use of the oracle. We can calculate the right hand side values which are all multiples of n, and then recover n itself by taking their GCD:

$$n = \gcd(n \cdot k_1, n \cdot k_2, \dots)$$

For instance, we can use the oracle to get the following values:

$$E(2) = 5015491228903933501466933977330839364265812322896587307873786047449411738906 8$$
$$E(3) = 7417716767886680651992933736668931393930001548923886454167963047600862721059 9$$
$$E(4) = 3417659032269469083397536494006342361506315984867578367502587339020697764547 6$$
$$E(6) = 6973283571185225304407518524850297071472962937338633619492778488634905382807 9$$

and then use these to get the value of n.

$$n = \gcd(E(2) \cdot E(2) - E(4), E(2) \cdot E(3) - E(6))$$

$$n = 76200708443433250012501342992033571586971760218934756930058661627867825188509$$

In this case, two terms were enough to reach the actual 256-bit value of n. If we would have gotten a larger n than expected, it would have required us to compute more $E(a), E(b), E(ab)$ pairs.

2.2 Finding e

Another important variable we have to find is e. In practice, this takes one of a few conventional values, especially including the Fermat Primes 3, 17, 65537. In this case, we are lucky and it is a common value. We can verify $e = 65537$ checks out:

$$2^{65537} \equiv 5015491 \dots 7389068 = E(2) \mod n$$

2.3 RSA internals

The strength of RSA stands in the fact that an attacker cannot feasibly factor n, which is semiprime, into the primes p, q. If one were able to do that, though, they would be able to calculate Euler's totient of n, then the decryption exponent d, and then use d and n to decrypt any message:

$$\varphi(n) = \operatorname{lcm}(p-1, q-1)$$

$$ed \equiv 1 \mod \varphi(n)$$

$$d \equiv e^{-1} \mod \varphi(n)$$

$$E(m)^d \equiv (m^e)^d \equiv m^{ed} \equiv m \mod n$$

2.4 Factoring n

As n is only 256 bits long, publicly available software such as YAFU or PARI/GP can factor it in a matter of minutes. Also, the organisers seem to have submitted the factorisation of n on factordb.com one hour before the second round started. Using either of these methods we obtain

$$p = 23208666403679275164626101821512345130 1$$
$$q = 32832868170035454673240472532058128680 9$$

and use the previous formulas to obtain

$$d = 8909286791397337049617644089239013721727832065716720078776080206046887604 73$$

2.5 Decrypting the message

We finally have all the pieces of the puzzle, so we can decrypt the given message.

$$m \equiv E(m)^d \mod n$$
$$m = 202010181600$$

And it looks like the timestamp 2020-10-18 16:00, which is a day before the start of the round. A plaintext which makes sense is always a good sign that our logic checks out. We can even verify that when put through the encryption oracle, 202010181600 gives the same encrypted value as the one in the problem statement.

A Python code (requires yafu)

```
import subprocess, re
from fractions import gcd
from Crypto.Util.number import inverse

Em = 715118966813248334583613928851843449333331598308638786001892120737775821781 73
E2 = 501549122890393350146693397733083936426581232289658730787378604744941173890 68
E3 = 741771676788668065199293373666893139393000154892388645416796304760086272105 99
E4 = 341765903226946908339753649400634236150631598486757836750258733902069776454 76
E6 = 697328357118522530440751852485029707147296293733863361949277848863490538280 79

n = gcd(E2*E2 - E4, E2*E3 - E6)
print(f"n = {n}")

e = 65537
assert pow(2, e, n) == E2

yafu_outp = subprocess.check_output(["./yafu-x64.exe", f"factor({n})"])
p, q = [int(x) for x in re.findall(r'P\d+ = (\d+)', yafu_outp.decode())]

print(f"p = {p}")
print(f"q = {q}")

assert p*q == n

L = (p-1)*(q-1)//gcd(p-1, q-1)
d = inverse(e, L)
print(f"d = {d}")

m = pow(Em, d, n)
print(f"m = {m}")
```