Being Eve

James Commons

Diffie - Hellman

What we know: $g=7$, $p=97$, $A = g^x \bmod p = 53$
$B = g^y \bmod p = 82$, shared secret $= B^x \bmod p = A^y \bmod p$

We need to find X or Y. I will look for X and check my work by finding Y.

$7^X \bmod 97 = 53$. I used brute force to find X by running this python code:

```
for x in range(1,100):
    if (7**x) % 97 == 53: print(i)
```

This found an x of 22 that works. This is the step that would have failed if the numbers were larger because I had to brute force to find X.

Anyway, $X = 22 \Rightarrow$ secret $= 82^{22} \bmod 97 = \boxed{65}$
Ascii "A".

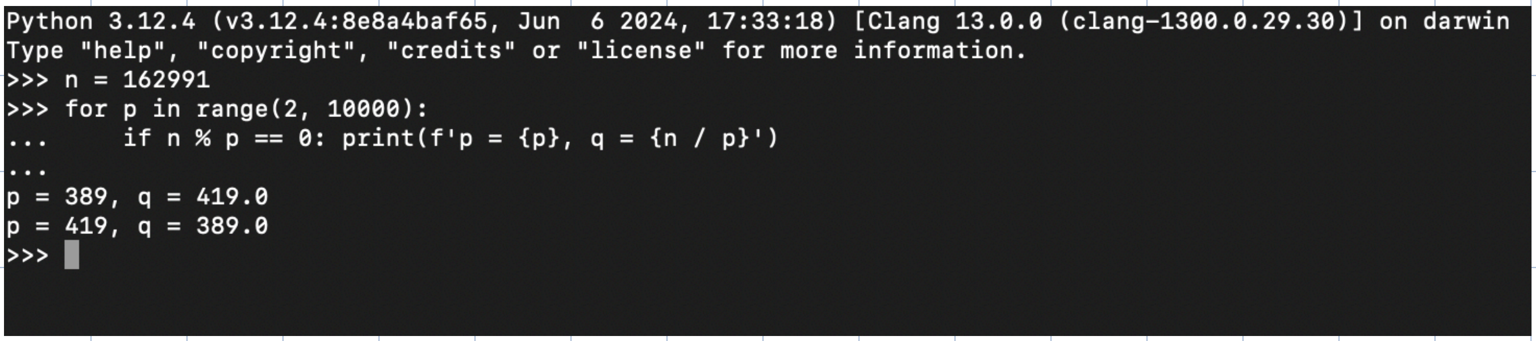
Doing the same steps for Y, I find $Y = 41$ and the secret is again 65, confirming I did things right.

RSA

Bob's public key: $(e = 13, n = 162991)$

All I need to do is decrypt the first number, then I will know d and can decrypt the rest.

We know $n = pq = 162991$. Now, since p and q are prime, there is only one non-trivial solution. Since the numbers are small enough, I can brute force it. Here is the code I used for this approach:



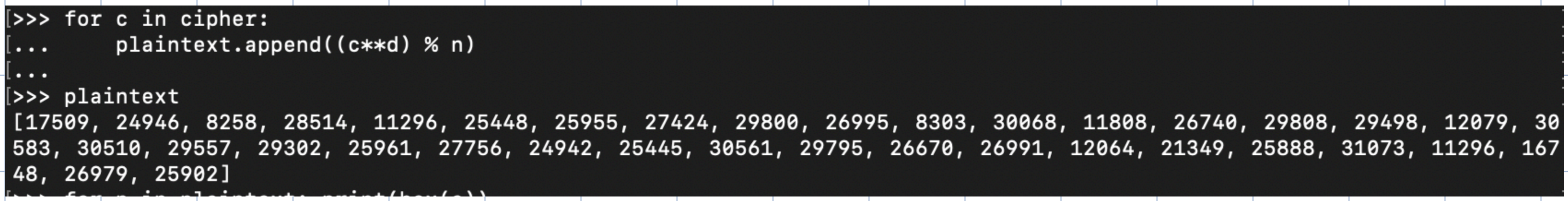
```
Python 3.12.4 (v3.12.4:8e8a4baf65, Jun  6 2024, 17:33:18) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> n = 162991
>>> for p in range(2, 10000):
...     if n % p == 0: print(f'p = {p}, q = {n / p}')
...
p = 389, q = 419.0
p = 419, q = 389.0
>>> 
```

So, $p = 389$, $q = 419$.

Now, Wolphram alpha has an RSA d calculator.

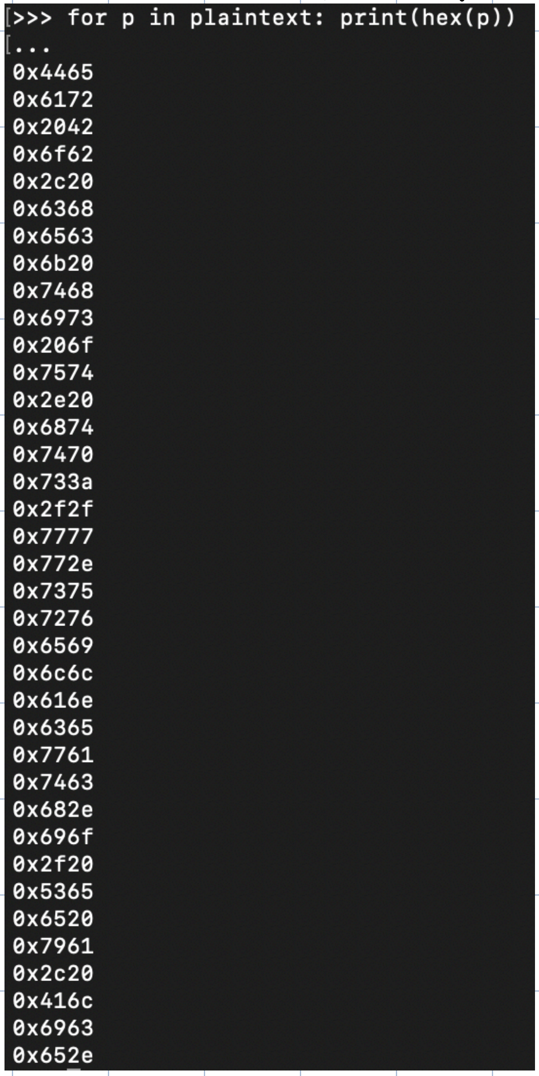
In practice, I could also use the extended Euclidean algorithm or brute force approach, but wolphram alpha works great. It found $d=124757$.

To decrypt x in ciphertext, use $m = x^d \bmod n$. Here is the code for that:



```
[>>> for c in cipher:
[...     plaintext.append((c**d) % n)
[...
[>>> plaintext
[17509, 24946, 8258, 28514, 11296, 25448, 25955, 27424, 29800, 26995, 8303, 30068, 11808, 26740, 29808, 29498, 12079, 30
583, 30510, 29557, 29302, 25961, 27756, 24942, 25445, 30561, 29795, 26670, 26991, 12064, 21349, 25888, 31073, 11296, 167
48, 26979, 25902]
```

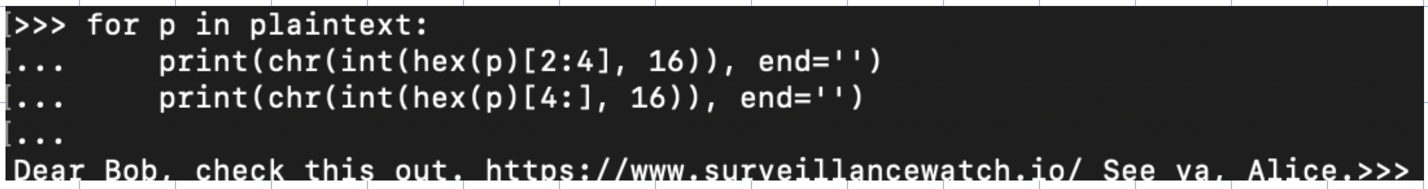
Or in hex:



```
[>>> for p in plaintext: print(hex(p))
[...
0x4465
0x6172
0x2042
0x6f62
0x2c20
0x6368
0x6563
0x6b20
0x7468
0x6973
0x206f
0x7574
0x2e20
0x6874
0x7470
0x733a
0x2f2f
0x7777
0x772e
0x7375
0x7276
0x6569
0x6c6c
0x616e
0x6365
0x7761
0x7463
0x682e
0x696f
0x2f20
0x5365
0x6520
0x7961
0x2c20
0x416c
0x6963
0x652e
```

There appears to be 2 bytes in each decoded number.

Running just a little bit more code so this is human readable, we get



```
>>> for p in plaintext:
...     print(chr(int(hex(p)[2:4], 16)), end='')
...     print(chr(int(hex(p)[4:], 16)), end='')
...
Dear Bob, check this out. https://www.surveillancewatch.io/ See ya, Alice.>>>
```

Decrypted message.

The part that would have been next to impossible to do with bigger numbers is finding p and q at the beginning. There is no known efficient way of finding these numbers, even though the algorithm I used technically works.

Why is this insecure anyway?
Each number was encrypted separately, similar to ECB mode of operation. This means two character sequences that occur more than once result in the same number in ciphertext. For example, 120780 (corresponding to a comma and a space), shows up twice in ciphertext.