Public Key Cryptography

This short write-up is an exploration of public-key cryptography. To use the method, one first generates a pair of keys called the public key and a corresponding private key. These have a symmetry property, so that if either one is used to encrypt a message, then the other one can be used to decrypt the message.

Suppose that Alice generates a public/private key pair and then distributes the public key in such a way that Bob is certain that he really knows Alice's public key.

Bob can encrypt a message with the public key and send it to Alice with the knowledge that only Alice can decrypt it using the private key. Alternatively, Alice can encrypt a message with her private key and send it to Bob. Successful decryption with Alice's public key proves that the message is from Alice.

The process of encryption and decryption is computationally expensive, and my understanding is that only short messages should be exchanged by this method in any case. Such a message would typically consist of yet another key which can be used to encode the main message.

We start with two prime numbers, p and q. I obtained mine from a list of primes (near the 50 millionth prime).

`http://primes.utm.edu/lists`

The prime numbers for real-world use are much larger than this.

Python code:

```
p = 961748941
q = 982451653
n=p*q
```

If you paste the lines above into the Python interpreter, you should be able to query the value of n like this:

```
>>> n
944871836856449473
>>> len(bin(n))
62
>>>
```

(I am showing the plain code to facilitate your copying and pasting into the interpreter to try it out).

I believe this means we can securely encode about 62 bits. Now compute

$$\phi(n) = (p-1)(q-1)$$

```
phi=(p-1)*(q-1)

>>> phi
944871834912248880
>>>
```

Now choose a number e

$$1 < e < \phi(n)$$

called the public key exponent, with the requirement that e should be coprime to $\phi(n)$ (they should have no common factor other than 1).

The easiest way to do that is to pick a prime number like

$$e = 2^{16} + 1 = 65537$$

I also confirmed that e is on a list of primes:

`http://primes.utm.edu/lists/small/10000.txt`

Just check that there is a remainder when dividing $\phi(n)$

```
>>> phi
944871834912248880
>>> e = 2**16 + 1
>>> e
65537
>>> phi % e
42186
>>>
```

The public key is (n, e).

The private key consists of n plus another number d which is computed from $\phi(n)$ and thus requires knowledge of p and q (below). That is why the process of breaking this method of encryption is described as being the same as the problem of finding two primes that can factor a large number (n, the product of the primes p and q). n is known because it is part of the public key.

encryption

We also need to generate a number representing our message m. ASCII-encoding is certainly a possible way to do this, though quite wasteful since (especially if we just use lowercase) most bytes do not ever appear in any message.

In this code sample, we convert the message ”hello world” into a binary number using the `ord` and `bin` functions. Because `bin` doesn't show leading zeroes, we add them back with `zfill`.

```
s = "hello world"
L = list()
for c in s:
    b = bin(ord(c))[2:].zfill(8)
    L.append(b)


>>> b = "".join(L)
>>> int(b,2)
126207244316550804821666916L
```

A more compact encoding might be achieved like this:

```
from string import lowercase as lc
D = dict(zip(lc,range(1,len(lc)+1)))
```

Don't forget the space:

```
D[" "] = 0
```

The `dict` assigns an integer for each lowercase letter. Now do:

```
s = "hello world"
m = 0
N = 27
for i,c in enumerate(s):
    m += D[c] * N**i
```

The number m is being formed from the integer values for each character of 'hello world'. For example, if the message were simply `"hel"` the value would be

$$8 + (5 \times 27) + (12 \times 27^2) = 8891$$

since "h" is the eighth letter, "e" is the fifth letter, and "l" is letter number 12.

The result can be viewed as:

```
>>> m
920321254041092
>>>
```

The number *is* our message. To read it, just reverse the process:

```
N = 27
m = 920321254041092
rD = dict(zip(D.values(),D.keys()))

while m:
    print rD[m % N],
    m /= N
```

Output:

```
h e l l o   w o r l d
>>>
```

While the above encoding could be viewed as a form of encryption, it is very weak. The encryption function we will use is

$$c = m^e \; mod \; n$$

```
m = 920321254041092
e = 65537
n = 944871836856449473
x = m**e

>>> len(str(x))
980692
>>> c = x % n
>>> c
448344912451359241L
```

The number c is our ciphertext. (The L on the end signifies a Python long, a type of number).

$x = m^e$ is a very large number!

It is much more efficient to do the mod operation at the same time as the exponentiation. The Python built-in function `pow` allows that as an option:

```
>>> pow(m,e,n)
448344912451359241L
```

decryption

There is one more number we need to decode the encrypted text. We will call this number d, and it is referred to as the exponent of the private key. The private key is (d,n), although just the d part is actually secret. Finding d is the only tricky part of the whole operation, but it only needs to be computed once for a given key pair.

d is called the modular multiplicative inverse of e (mod $\phi(n)$).

What this means is that we want d such that

$$d \times e = 1 \ (mod \ \phi(n))$$

Substituting the known values for e and $\phi(n)$

$$d \times 65537 = 1 \ (mod \ 944871834912248880)$$

I found a simple implementation for computing d here:

`stackoverflow.com/questions/4798654`

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b \% a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception("modular inverse does not exist")
    else:
        return x % m
```

I have not figured out how it works, but it is easy to show that it does. I saved the code in a file `mod.py`. Let's try it out:

```
from mod import modinv
e = 65537
phi = 944871834912248880
d = modinv(e,phi)
```

Output:

```
>>> d
8578341116816273
>>> d*e % phi
1L
>>>
```

So now we are finally ready to decrypt:

```
c = 448344912451359241
n = 944871836856449473
d = 8578341116816273
p = pow(c,d,n)
```

```
>>> p 920321254041092L
>>>
```

Recall

```
>>> m = 920321254041092
```

We have successfully generated a key pair, and used it to encrypt and decrypt a simple message. Now we need to show that we can also encrypt with private key, and decrypt with public one:

```
m = 920321254041092
d = 8578341116816273
n = 944871836856449473
c = pow(m,d,n)
```

```
>>> c
461000660869754451L
```

```
e = 65537
p = pow(c,e,n)
```

```
>>> p
920321254041092L
```

We have again recovered our plaintext message: `p = m`.