***Setting the Clock Back-***

This might allow all kinds of mischief. The machine mistakenly believes it lives in the past. Maybe an attacker once had access to some data because he was a temporary employee, but that access has now expired. With the wrong time on the clock, a computer might now allow this ex-employee access to the sensitive data.

***Stopping the Clock***

But if the clock is stopped, time appears to stand still. Things might not get done. And many systems behave in unpredictable ways. The simple problems are things like getting the wrong time on audit logs and reports. The exact time of a transaction can have large financial consequences and sending out formal paperwork with the wrong date and time on it can lead to serious complications.

***Setting the Clock Forward***

This leads to simple denial-of-service attacks. With the clock set four years in the future, all credit card transactions are suddenly refused because all the cards have expired. You cannot book online airline tickets either, because there is no airline schedule out yet for those dates. Substantial bidding at eBay auctions happens in the last seconds. If you can move eBay’s clock forward just a little bit, you cut out many of the other bidders and can obtain the item at a cheaper price.

Some examples of attacks that can be carried out against systems that rely on time via NTP are

* Certificate Attacks
* Authentication Attacks
* DNSSEC Attacks
* NTP Amplification

#Encoding: UTF-8

import random

def gcd(a, b):

'''Return the greatest common divisor (gcd) of two numbers.

Use the recursive variant of the extended Euclidean algorithm

to compute the GCD of a and b. Also find two numbers x and y

that satisfy ax + by = gcd(a,b).

'''

if b == 0:

return (a, 1, 0)

(d\_, x\_, y\_) = gcd(b, a % b)

(d, x, y) = (d\_, y\_, x\_ - (a // b) \* y\_)

return (d, x, y)

#Fermat's primality test used with base 2. This will return reliable prime numbers up to prime<=340.

def isPrime(number):

if ( (2 \*\* (number -1) ) % number ) == 1:

return (True )# and isPrime(number,base-1))

else:

return False

def gen\_prime(max):

for i in xrange(max,2,-1):

if(isPrime(i)):

return i

def gen\_privkey(p, q, e):

''' Return the private key exponent d.

Compute the private exponent d using the extended Euclidean algorithm.

d is the multiplicative inverse of e % ((p - 1) \* (q - 1)),

so it satisfies d \* e == 1 % ((p - 1) \* (q - 1)).

'''

phi\_n = (p - 1) \* (q - 1)

(x, d, y) = gcd(e, phi\_n)

if d < 0: d += phi\_n

return d

def pubkey\_encrypt(text, pubkey, factor):

return text \*\* pubkey % factor

def privkey\_decrypt(chiffre, privkey, factor):

return chiffre \*\* privkey % factor

print "Generating prime p"

prime\_p = gen\_prime(random.randint(10,50))

print ">>Prime p:" + str(prime\_p)

print "Generating prime q"

prime\_q = gen\_prime(random.randint(10,50))

print ">>Prime q:" + str(prime\_q)

assert prime\_p != prime\_q

print "Calculating factor n"

factor\_n = prime\_p\*prime\_q

print ">>Factor n:" + str(factor\_n)

print "Calculating Phi(N)"

phi\_n = (prime\_p-1)\*(prime\_q-1)

print ">>Phi(N):" + str(phi\_n)

# e doesn't have to be choosen at random. any prime number < phi\_n will work.

e = 17

print ">>Public exponent e:" + str(e)

assert e < phi\_n

print "Generating pubkey d"

privkey\_d = gen\_privkey(prime\_p, prime\_q, e)

print ">>Privkey d:" + str(privkey\_d)

print "Generating text"

text\_clear = random.randint(0,100)

print ">>Text:" + str(text\_clear)

print "Encrypting"

text\_encrypted = pubkey\_encrypt(text\_clear, e, factor\_n)

print ">>Text (encrypted):" + str(text\_encrypted)

print "Decrypting"

text\_decrypted = privkey\_decrypt(text\_encrypted, privkey\_d, factor\_n)

print ">>Text (decrypted):" + str(text\_decrypted)