Wieners Attack ENG

February 7, 2020

1 Wiener's attack

RSA decryption (or signature) creation tends to be relatively energy and time expensive, since we can't just pick a nice private exponent d with low Hamming weight (it would be easily guessable). So one may experience an urge to peek such public exponent e, that d bitlength is much less than the modulus N, say if N is 2048, than d is around 500 bits. Intuition says, that since d is hard to guess (500 bits), you should feel safe with this decision. Unfortunately, this is and extremely dangerous mistake to make. ## Theorem (M. Wiener) Let $N = pq$ with $q < p < 2q$. Let $d < \frac{1}{3}N^{\frac{1}{4}}$. Given (N, e) with $ed = 1 mod \varphi(N)$, Marvin can efficiently recover d.

1.0.1 Proof (taken from Twenty Years of Attacks on the RSA Cryptosystem by Dan Boneh)

The proof is based on approximations using continued fractions. Since $ed = 1 mod \varphi(N)$ there exists a k such that $ed - k\varphi(N) = 1$. Therefore,

$$|\frac{e}{\varphi(N)} - \frac{k}{d}| = \frac{1}{d\varphi(N)}$$

Hence, $\frac{k}{d}$ is an approximation of $\frac{e}{\varphi(N)}$. Although Marvin does not know $\varphi(N)$, he may use N to approximate it. Indeed, since $\varphi(N) = N-p-q+1$ and $p+q-1 < 3\sqrt{N}$, we have $|N-\varphi(N)| < 3\sqrt{N}$

Using N in place of $\varphi(N)$ we obtain:

$$|\frac{e}{N} - \frac{k}{d}| = |\frac{ed - k\varphi(N) - kN + k\varphi(N)}{Nd}| = |\frac{1 - k(N - \varphi(N))}{Nd}| \leq |\frac{3k\sqrt(N)}{Nd}| = \frac{3k}{d\sqrt{N}}$$

Now, $k\varphi(N) = ed - 1 < ed$. Since $e < N$, we see that $k < d < \frac{1}{3}N^{\frac{1}{4}}$. Hence we obtain:

$$|\frac{e}{N} - \frac{k}{d}| \leq \frac{1}{dN^{\frac{1}{4}}} < \frac{1}{2d^2}$$

This is a classic approximation relation. The numbber of fractions $\frac{k}{d}$ with $d < N$ approximating $\frac{e}{N}$ so closely is bounded by $log_2 N$ In fact, all such fractions are obtained as convergents of the continued fraction expansion of $\frac{e}{N}$ All one has to do is compute the $logN$ convergents of the continued fraction for $\frac{e}{N}$. One of these will equal $\frac{k}{d}$. Since $ed - k\varphi(N) = 1$, we have $gcd(k, d) = 1$ and hence $\frac{k}{d}$ is a reduced fraction. This is a linear-time algorithm for recovering the secret key d. **Q.E.D**

1.1 Continued fractions and convergents

Continued fraction is an expression obtained through an iterative process of representing a number as a sum of its integer part and the reciprocal of another number, then writing this other number as the sum of its integer part and another reciprocal and so on. Example: For real $x > 0$ and integers $a_i > 0$, for $i = 1, ..., n$

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \frac{1}{a_n}}}}$$

is a continued fraction. The ingegers a_0, a_1, etc., are called the *coefficients* or *terms* of the continued fraction. There are many ways to abbreviate continued fractions. We'll be using this one:

$$x = [a_0; a_1, a_2, ...]$$

A continued fraction can be approximated by its initial segments. Such approximations are called *convergents*. Obviously, for a rational x the number of such convergents is finite. The first four convergents of a continued fraction are:

$$\frac{a_0}{1}, \frac{a_1 a_0 + 1}{a_1}, \frac{a_2(a_1 a_0 + 1) + a_0}{a_2 a_1 + 1}, \frac{a_3(a_2(a_1 a_0 + 1) + a_0) + (a_1 a_0 + 1)}{a_3(a_2 a_1 + 1) + a_1}$$

If there are successive convergents, they can be calculated recusively. h_i are numerators and k_i are denominators. Then the series can be calculated this way:

$$\frac{h_i}{k_i} = \frac{a_i h_{i-1} + h_{i-2}}{a_i k_{i-1} + k_{i-2}}$$

Now implement the algorithm for finding coefficients and convergents for any given rational number.

Hint: you can use Euclidian algorithm for finding coefficients

[ ]:

Now that you've implemented the algorithm to find convergents, let's see how to use them to find p and q. Since $\frac{k}{d}$ is a reduced fraction, the denominator of one of the convergents is the d itself. This is enough to decrypt, but we can even factor the N. We also know k, and $\varphi(N) = \frac{ed-1}{k}$, so we can calculate $\varphi(N)$.

$$N - \varphi(N) = pq - (p-1)(q-1) = pq - pq + p + q - 1 = p + q - 1$$

$$q = N - \varphi(N) - p + 1$$

$$N = pq = p(N - \varphi(N) - p + 1) = pN - p\varphi(N) - p^2 + p$$

$$p^2 - p(N - \varphi(N) + 1) + N = 0$$

We have a nice quadratic equation, the roots of which are p and q

Now let's try to apply this knowledge to a vulnerable server. The concept is the same as in the blinding task. You need to provide a valid signature for 'flag' to get the flag. Steps: 1. Find d through the cunning use of continued fractions 2. Find p and q 3. Sign message and send to server 4. Profit

[1]:
```
import socket
import re
from Crypto.Util.number import inverse,long_to_bytes,bytes_to_long
class VulnServerClient:
    def __init__(self,show=True):
        """Initialization, connecting to server"""
        self.s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.s.connect(('cryptotraining.zone',1338))
        if show:
            print (self.recv_until().decode())
    def recv_until(self,symb=b'\n>'):
        """Receive messages from server, by default till new prompt"""
        data=b''
        while True:

            data+=self.s.recv(1)
            if data[-len(symb):]==symb:
                break
        return data
    def get_public_key(self,show=True):
        """Receive public key from the server"""
        self.s.sendall('public\n'.encode())
        response=self.recv_until().decode()
        if show:
            print (response)
        e=int(re.search('(?<=e: )\d+',response).group(0))
        N=int(re.search('(?<=N: )\d+',response).group(0))
        self.num_len=len(long_to_bytes(N))
        self.e,self.N=e,N
        return (e,N)

    def checkSignatureNumber(self,c,show=True):
        """Check if this number is a valid signature for 'flag'"""
        try:
            num_len=self.num_len
        except KeyError:
            print ('You need to get the public key from the server first')
            return
        signature_bytes=long_to_bytes(c,num_len)
        self.checkSignatureBytes(signature_bytes,show)

    def checkSignatureBytes(self,c,show=True):
        """Check if these bytes are a valid signature for 'flag'"""
        try:
            num_len=self.num_len
        except KeyError:
            print ('You need to get the public key from the server first')
```

```
            return
        if len(c)>num_len:
            print ("The message is too long")
            return

        hex_c=c.hex().encode()
        self.s.sendall(b'flag '+hex_c+b'\n',)
        response=self.recv_until(b'\n').decode()

        if show:
            print (response)

        if response.find('Wrong')!=-1:
            print('Wrong signature')
            x=self.recv_until()
            if show:
                print (x)
            return
        flag=re.search('CRYPTOTRAINING\{.*\}',response).group(0)
        print ('FLAG: ',flag)
    def setPrivateKey(self,p,q):
        """Set private key"""
        self.p=p
        self.q=q
        self.d=inverse(self.e,(p-1)*(q-1))

    def signMessageBytes(self,m):
        """Sign message after finding private key"""
        try:
            num_len=self.num_len
        except KeyError:
            print ('You need to get the public key from the server first')
            return
        if len(m)>num_len:
            print('m too long')
        if len(m)<num_len:
            m=bytes([0x0]*(num_len-len(m))) + m
        signature_bytes=long_to_bytes(pow(bytes_to_long(m),self.d,self.N))
        return signature_bytes

    def __del__(self):
        self.s.close()
```

[2]:
```
vs=VulnServerClient()
(e,N)=vs.get_public_key()
```

```
Welcome to the Wiener attack task
```

```
 Private exponent d is just 500 bits, so you should be able to find it
Available commands:
help - print this help
public - show public key
flag <hex(signature(b'flag'))> - print flag
quit - quit
>
e: 883955104397844360839802589678079320200301053675860631429632241079629322700900637792870473311552753056833515750354593566814028068767764438482882777434179174167531101705349192056441564161813881940662103858990879528064249455962302439810906861229840688299918154674780937236275392436724283337231900580915064209944590829593179768699333704428744236644694463302339528604487059166123052119894425485654008873775391149102634853875531641579809554358787031415968474616444976201252313903301798430556619873714503970183426633254430547742575886769977055864996939801428411328381096399767826718950452118659784154891195826991661922961 5
N: 212563224300895988543387006892002719036754137409523146508775633055952002982147958402794086106494975351559198861994940940941139463270388550283531131123758958799623003985099830415299465089729855511075800086427546384784838006984961812360327544778630995608776463045786564342649998465076640576496290411556880916730877373395798379615001531403504381828335354290650587041012836487617168294034288081060340011418775963432932239367388470383293641635502826193087609782306888074314016493443556910165719968210776833910625874738412965611958236748625848488151784709690389684573576432874392053703896310262986981095078473647234183097095 
>
```

[ ]:

[ ]: