RSA Blinding ENG

February 7, 2020

1 RSA Blinding Attack

1.1 Intro

RSA (named after its creators Ronald Linn Rivest, Adi Shamir and Leonard Adleman) is an example of asymmetric cryptosystem, that can be used for secure communications and signing data. The basic principle behind it is as follows.

RSA uses the multiplicative group modulo $N = pq$, where p and q are prime numbers. The order of the multiplicative group (effectively the number of elements in this group) can be calculated with Euler's totient function: $\varphi(N) = (p-1)(q-1)$. We just cross out the numbers less than N that are multiples of p or q. All the other numbers are in the multiplicative group, since they are coprime with N. As we know, if we exponentiate any member of a group to the order of the group, the result is group's identity: $a^{\varphi(N)} = 1, a \in Z_N^*$. So two numbers, e (public exponent) and d (private exponent) such that $ed = 1 mod N$ are chosen. The pair of numbers (e, N) becomes the public key and (d, N) becomes the private key. Calculating d from public information is considered a hard problem (unless certain mistakes were made). You would to have to factor N to get p and q.

Given message $M, M < N$, public key (e, N) and private key (d, N) the encryption and decryption process is as follows:

Encryption $C = M^e mod N$

Decryption

$M = C^d mod N$

It's quite easy to check soundness:

$C^d mod N = M^{ed} mod N = M^{edmod\varphi(N)} mod N = M^1 mod N = M mod N$

1.2 Preparation

Let's try to work with RSA for a bit. If you haven't yet, install Pycryptodome. On Linux and Windows this should work (You obviously have to install python 3 first and pip, but I hope you know how to do that):

[1]:
```
!python3 -m pip install pycryptodome
```

```
Collecting pycryptodome
  Using cached https://files.pythonhosted.org/packages/54/e4/72132c31a4cedc58848
615502c06cedcce1e1ff703b4c506a7171f005a75/pycryptodome-3.9.6-cp36-cp36m-manylinu
x1_x86_64.whl
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.9.6
```

You'll have to restart jupyter kernel after installation (circly arrow near "Run"). If you encounter problems, you can follow this installation guide: Pycryptodome installation.

1.3 Primitive RSA

So let's try to implement RSA. Let's use the public exponent $e = 65537$. This constant is usually chosen nowadays, because it wraps the modulus even if $M = 2$ and it has a nice binary representation $65537_{10} = 10000000000000001_2$ which allows for efficient exponentiation using the "Square and multiply" method.

First, generate the p and q. getStrongPrime function lets you choose the number of bits in your prime and checks that $gcd(p-1, e) = 1$

[2]:
```
try:
    from Crypto.Util.number import getStrongPrime, inverse,bytes_to_long,␣
 ↪long_to_bytes
except ImportError:
    print ("Pycryptodome not installed")
```

[3]:
```
e=65537
p=getStrongPrime(1024,e=e)
q=getStrongPrime(1024,e=e)
```

[4]:
```
N=p*q
phi=(p-1)*(q-1)
d=inverse(e,phi)
public_key=(e,N)
private_key=(d,N)
```

Ok, we've generated the keys, let's encrypt a message, decrypt the ciphertext and check if it is the same message

[5]:
```
M=bytes_to_long(b'Hello, RSA!')
C=pow(M,e,N)
print ('C:',hex(C))
M1=pow(C,d,N)
assert M1==M
print ('M1:',long_to_bytes(M1))
```

```
C: 0x57d826694f86a3120e40f5a86bfbd451a91e40886345fcbeb360582f9334bfd5d0734a0f2d1
1068da5aa851fb07d265ac3cddf47092a9d6e8049801bf721db5a8ae2a78dbae899212a14c22e9a7
```

```
128f54158b143f22410997184c75b1945a30b940d921e43e05a401bffb3c356ed2134d503c4a112b
9b3782e3c85a9b5985d7b836a6b9fdffe941bfaac5555583716e8667c9ba8076cc9ad1063428abde
01e2639f07afd55bff37c58d01dc4300563a8d01bc253689024b5d124639723f67bc3a69d867cb8c
89a4f3f373dac4d054931e9c452f5fcbba384d78a219d515fa8a0803f3095531b5e865e0ad843029
6e6e1b3ee60a193376624db9561395abbe5f9
M1: b'Hello, RSA!'
```

Creating a signature is inverse operation to encryption.

$$Sign(M) \equiv Dec(M), Check(S) \equiv Enc(S)$$

This way anyone with a public key can check the validity of the signature, but only the entity holding the private key can produce signatures. Congratulations. You know how to encrypt and create signatures with RSA. Now let's explore one of RSA's interesting properties. ## RSA Blinding RSA is homomorphic encryption under multiplication. Homomorphism is a structure-preserving map between two algebraic structures of the same type. (If you didn't get anything from that, that's ok. I also didn't the first time I heard that). What this means is that if you have to elements (x, y) and you put them through a homomorphic function, they will relate the same to each other in the new group/ring/field, etc. as they were in the original one:

$$\varphi(x \cdot y) = \varphi(x) \times \varphi(y)$$

This translates to

$$Enc(M_1 \cdot M_2) = Enc(M_1) \times Enc(M_2) =$$

The same is true for decryptions:

$$Dec(C_1 \times C_2) = Dec(C_1) \cdot Dec(C_2)$$

Let's try this in python

[6]:
```python
class BasicRSA:
    def __init__(self, e,p,q):
        self.e=e
        self.p=p
        self.q=q
        self.N=p*q
        self.d=inverse(e,(p-1)*(q-1))

    def encryptNumber(self, m):
        return pow(m,self.e,self.N)

    def decryptNumber(self, c):
        return pow(c,self.d,self.N)

brsa=BasicRSA(e,p,q) #we created these parameters earlier
m1=2
m2=3
m3=m1*m2
c1=brsa.encryptNumber(m1)
c2=brsa.encryptNumber(m2)
```

```
print ('c1:',c1)
print ('c2:',c2)
c3=(c1*c2)%brsa.N
print('c3:',c3)
m3_dec=brsa.decryptNumber(c3)
print ('m3: %d, m3_dec: %d'%(m3,m3_dec))
assert m3_dec==m3
```

```
c1: 1101435848518319984671476776473258854436356021673498699775175983898089624703
6812118141673755964065645338643639675248246068562288738065082417407730360184076 2
1499058879197518189229604272156382780573174095112460970657990131865375845677533 0
4737104914854313285838443915403587184613246372416755129400996430397490205996466 5
0211762103144205410613508156429764270484909050485216692928577495624223260765599 3
0419030561896483657967885697456922750945192469838610629447727733443362238781543 9
3660089629214145415068764690895512051899607175333513925431156599517411530053903 7
4003996592787607476215857261150299842299244927932785444540546
c2: 2226991733155058768106286972533617684870026735615018555763443274196304773050
3809242923347246719638703856668622377030750771723279059943151242898428084220650 7
1308745883874053441659066989363118783083747337047939425548034290237240054980475 3
5441668396435657003755547059226529254674085482775770916733973361013681270191750 0
7277668743496322415140615497487853756415578742443831527343497997377644408976354 6
0512443898630248401732191255359582600045680626542092273396115068722559628047764 6
2171594391830621908278931433086261493544725991004125188487732616097860268746180 8
9497180067645889625732747409915387100164530135060800138340313
c3: 2250405393745889240452722405071496942856944853732323747083382347314402245854
4344556207521869104783295633660944456502101463815947993350430308616085701383812 5
1541759895315936933567274972838961656143334750501743752343573741870523285203848 2
7984915645234414478241426688392000945593543998121065424805376743935020667823941 1
0241802386949238388571371607041040551279499933746305239402429060556217536316265 5
0564074065719005951111976546701407683908962128386162410479772532952990678205207 6
5165045822686032951693238667635282722191734954706031921779799705130506500167131 2
8074934226820960534119564475133998551385909075487395766058762
m3: 6, m3_dec: 6
```

1.4 Attacking the server

Now try to apply this knowledge to a vulnerable server. You can connect to it with `nc cryptotraining.zone 1337` or by using python sockets.

[7]:
```
import socket
import re
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',1337))
        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))
        return (e,N)

    def signBytes(self,m,show=True):
        """Get a signature for chosen byte message from the server"""
        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 ("The message is too long")
            return
        if len(m)<num_len:
            m=bytes((num_len-len(m))*[0x0])+m
        hex_m=m.hex().encode()
        self.s.sendall(b'sign '+hex_m+b'\n')
        response=self.recv_until().decode()
        if show:
            print (response)
        if response.find('flag')!=-1:
            print('You tried to submit \'flag\'')
            return None
        signature_hex=re.search('(?<=Signature: )[0-9a-f]+',response).group(0)
        signature_bytes=bytes.fromhex(signature_hex)
        return bytes_to_long(signature_bytes)


    def signNumber(self,m,show=True):
```

```
        """Get a signature for chosen number from the server"""
        try:
            num_len=self.num_len
        except KeyError:
            print ('You need to get the public key from the server first')
            return
        return self.signBytes(long_to_bytes(m,num_len),show)

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

[8]:
```
vs=VulnServerClient()
(e,N)=vs.get_public_key()
```

```
Welcome to RSA blinding task
Available commands:
help - print this help
public - show public key
sign <hex(data)> - sign data
flag <hex(signature(b'flag'))> - print flag
quit - quit
>
e: 65537
N: 20159717663186764200842482638329142432479376755681286432561400011207751568770
23937873504239055098886463647821209788938254180637863281345152201173477839435246
47506954302364591564396569321085369361070927857591871209155591733213020275252290
18106368725032056109022369913503577023942696069608771010384365856481001383579432
84411223121576763032862701509742254008778946240450869708632121399086803127321961
48979014368449994422593874530212706423955318848486976509334781242540719122324457
08062597679170291021925633789812405697682134528381868778865376836541179591638312
15247213631375725238476129368433608284013777398457594745906 1
>
```

You can sign messages with signNumber and signBytes methods.

You can check signatures with checkSignatureNumber and checkSignatureBytes methods.

Your goal is to get the valid signature for message 'flag'

Remember that RSA is homomorphic and solve the task.

Good luck!

[ ]: