Cryptographie - PrimalRSAEncryption - 200 points

Anass ABOUZI, Kévin DUVERGER

Table des matières

1 Résolution PrimalRSAEncryption :

La première chose que l'on peut faire comme dans tous les autres challenges est de se connecter au serveur pour voir ce qu'il retourne :

```
C:\Users\keked>ncat 146.59.227.136 23422
libnsock ssl_init_helper(): OpenSSL legacy provider failed to load.

Welcome to the StrongRSA authentication service.
This is an encryption-based authentication : I give you a ciphertext, you give me the plaintext to show me you kn
ow the private key.

=============================== PART 1 ===============================
Our brilliant scientists have found a way to make RSA probabilistic :
        Enc(pk = (n, e), m) :
                r = random 1024 bits prime
                return m^e mod (n * r)
This should OBVIOUSLY make it STRONGER, furthermore the size of the modulus increases !

You can now request a ciphertext (enter 1) or try to authenticate (enter 2) : 1
        n = 24084554593387389298391492442232649963608779301150697196422935828335991486033626143440779806044886666
29608501831642278643550041297028016104714863511809751282672989279699580734239235802289776951934809482530026703734
81087742423587903847558439868613128790825717775929414148284906901385589051756433498366529680335554696252164501392
23354704002429872907538433181465191772737950363727037412965836487480793051600346346145513020197876483288970326578
54666171733964449640332390593704419284346483678870864182382074140477621839380861522095320904488335977035321594091
0014833267626881157009543140963710644216308506323068745237286023
        e = 65537
        r = 16894180444854883774284395027862797954509027157039494809395773538407650639930380946457003757628661402
98901425264789435807311151763121129483534093790504683680588807517787178922242838933795310482136751451964100010877
10657198502539623899455651642570583505107648879620438761918012908509936229751755749832748264413
        c = 26160333160684055190322627399944784036505336651305194609854611166138253866021043384656733976516535038
02577206216271015505274052416644448335274647259822927055749350603708131041706437044796286671921675248035572848456
67548101913772863631030270126834241695307312130129220959837107959297439596145686217382276937287488466467340196960
18362905014682406993391559719833188755593239304989512861748614906321121698259704782442506469982242213621469917134
46395921000248968878369384620769550620175230011044078691014150654875917301725243267126296287465817224537112825381
90967814278566040048230843354660796012434777684266564501967114490952756140082669737252042149817722537100720198567
61443219187886589939556128360176224045428707657911344865142436324446146786449546321089834837992172355455146933798
14315002704853385257305660854875999247107292716025861181690545564253224743274800694424448080821101493756259196724
203159677900374491871556537740675

You can now request a ciphertext (enter 1) or try to authenticate (enter 2) : ^C
```

Comme vous pouvez le voir, le serveur nous annonce qu'il utilise une autre version de RSA, probabiliste. Dans un RSA normal, le chiffré se calcule avec $m^e \pmod n$ mais ici comme vous pouvez le voir on calcule plutôt $c = m^e \pmod{n \times r}$. Cette valeur de r est un nombre premier aléatoire sur 1024 bits qui est re-généré à chaque fois qu'on fait un chiffrement.
Pour l'exploiter, remarquez que l'on peut faire un déchiffrement RSA uniquement sur r. L'idée est donc de calculer $d = e^{-1} \pmod{\varphi(r) = r - 1}$, puis $c^d \pmod r = m^{ed} \pmod r = m^{1+k(r-1)} \pmod r = m^1 \pmod r = m \pmod r$! Donc si le message est plus petit que le nombre de bits de r, on peut le récupérer un seul coup en faisant ce calcul ! Nous pouvons donc essayer sur l'exemple de la capture pour voir ce que l'on obtient :

```
d = pow(e, -1, r - 1) : 18905644045739920594565169771937342294943193183961373717458626902
46451772178302544536912973716352636366044074780958194944965437391203497815316392855877038
94188600551374813772612859617322349315287409175713027899961437044706626428674135008887978
79253847848688479532726519064142494636798637548245672967535526057
token = pow(c, d, r) : 250045117590954933880941577372838850861896818785633362142831261543
07112240247251992909768671122557529013999528544099201250384928163792502761373279488865701
8841179302892923862612067390
token_bytes = token.to_bytes(128, byteorder="big")
b'\x00...\x00<BEGIN>yqyptnlrdiruabkksfnzlejxblebjhxawkbccgfvkguieqmtfmotlmnoizzugelv<END>'
```

Comme vous pouvez le voir, nous avons bien le token et nous pouvons l'envoyer au serveur avec le choix 2 !

Malheureusement, ce n'est pas la fin du challenge comme vous pouvez le voir sur la capture suivante :

```
Yay ! You passed the first level !!
=============================== PART 2 ===============================
HA ! Our brilliant scientists knew your plan and have planned a countermeasure !
The new scheme is the following one :
        Enc(pk = (n, e), m) :
                r = random 32 bits prime
                return m^e mod (n * r)
Good luck trying to break it now (also the tag is much longer), AHAHAHAHAHAH !

You can now request a ciphertext (enter 1) or try to authenticate (enter 2) : 1
        n = 15241534178523592055008378683798579195499540106527444806605836451720712805549894019254453954098146428
76671963508002940749841342225138140297525903163270388901270459252516721774904242486217019783078789011809422860315
77594739880521238153847032973774972932061121596141833892957234339134579891015891453848406659847692380006303177067
22058122688408701810713867976274245114467982338540997529698570723512593045108455159472444656003907851488169968918
48016944441189738804424335699055120512857229591052489706682523928957853868127538622079189958982185419154605519162
0737307317381745260956120625945749131678567581836205680985292561
        e = 65537
        r = 3255220369
        c = 26664294045824920688246903501562796321555031138529366151487537988488611330390766092404275214585132910
33954419504962796695222805110972609212763276991775112385051112062758583669664035653341284060593451805855318110184
13760785815734000753543301712292328414936025857772033085932268296054997633257968378608338878782341256178405846755
01070771006224697219111250724289652338641178792120558848868522331005307003859377674367268989782357316401320076963
64460140653923882667768924044528322203595746799883135097701496496144381551399835886523852280523768087629500618943
2242282215988288858558112350186944016609585904937071599066452366921359078

You can now request a ciphertext (enter 1) or try to authenticate (enter 2) : |
```

Cette fois-ci, la valeur de r est beaucoup plus petite (environ 32 bits), et donc on ne pourra pas récupérer autant d'informations que dans le cas précédent. En revanche, on peut récupérer plusieurs couples (r_i, c_i) comme vous pouvez le voir, et on pourrait "combiner" tous les $m_i = c_i^{d_i} \pmod{r_i}$ pour obtenir la totalité du tag. Cette combinaison se fera via le théorème chinois des restes (CRT) qui fonctionnera très bien ici car les r_i sont des nombres premiers (et on a peu de chances de tomber 2 fois sur le même).
Cette fois-ci, l'idée est la suivante :

```
# Recuperation
ri, ci = [], []
Pour i allant de 0 à 64 : # 64 = 2048 / 32
    r, c = recupererRCServer()
    ri = ri + [ r ]
    ci = ci + [ c ]

# Calcul des dechiffrements
mi = []
Pour i allant de 0 à 64 :
    di = pow(e, -1, ri[i] - 1)
    m  = pow(ci[i], d, ri[i])
    mi = mi + [ m ]

# CRT
m = CRT(mi, ri)
```

Ce qui permet d'obtenir le second tag d'authentification et donc le flag !

Voici le code de la solution complète :

```python
import socket

server = "146.59.227.136"
port = 23422

#Connecting to the server
maSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try :
    maSocket.connect((server, port))
except Exception as e :
    print("Probleme de connexion", e.args)
    sys.exit(1)

# Getting the starting screen
data = maSocket.recv(16384).decode("utf-8")
maSocket.sendall(b"1\n")

# Now we can parse what the server sends us
data = maSocket.recv(16384).decode("utf-8")
lines = data.split("\n")
n = int(lines[0].strip()[3:])
e = int(lines[1].strip()[3:])
r = int(lines[2].strip()[3:])
c = int(lines[3].strip()[3:])

# Computing the authentication tag
exponent = pow(e, -1, r - 1)
token = pow(c, exponent, r) # decrypting modulo r
tokenBytes = token.to_bytes(128, byteorder="big")
while tokenBytes[0] == 0 :
    tokenBytes = tokenBytes[1:]
print("Token 1 : " + str(tokenBytes))
maSocket.sendall(b"2\n")
data = maSocket.recv(16384).decode("utf-8")
maSocket.sendall(tokenBytes + b"\n")

# Getting the response from the server
data = maSocket.recv(16384).decode("utf-8")
# We are going to need 2048 / 32 = 64 samples (n and e are always the same)
n, e = None, None
samples = []
while len(samples) < 64 :
    maSocket.sendall(b"1\n")
    data = maSocket.recv(16384)
    lines = data.split(b"\n")
    if n == None : n = int(lines[0].strip()[3:])
    if e == None : e = int(lines[1].strip()[3:])
    r = int(lines[2].strip()[3:])
    c = int(lines[3].strip()[3:])
    if r not in [ samples[i][0] for i in range(0, len(samples)) ] :
        samples.append([ r, c ])
```

```
# Now that we have the samples, we can decrypt all of them
decryptions = []
for i in range(0, len(samples)) :
    exponent  = pow(e, -1, samples[i][0] - 1)
    decrypted = pow(samples[i][1], exponent, samples[i][0])
    decryptions.append([ samples[i][0], decrypted ])
# Now doing CRT to find the plaintext
R = 1
for i in range(0, len(decryptions)) :
    R *= decryptions[i][0]
sumRi, sumRiAi = 0, 0
for i in range(0, len(decryptions)) :
    Ri = R // decryptions[i][0]
    sumRi += Ri
    sumRiAi += Ri * decryptions[i][1]
token = (sumRiAi * pow(sumRi, -1, R)) % R
tokenBytes = token.to_bytes(256, byteorder="big")
while tokenBytes[0] == 0 :
    tokenBytes = tokenBytes[1:]
print("Token 2 : " + str(tokenBytes))
# Sending the second token
maSocket.sendall(b"2\n")
data = maSocket.recv(16384).decode("utf-8")
maSocket.sendall(tokenBytes + b"\n")
data = maSocket.recv(16384).decode("utf-8")
print(data)

maSocket.close()
```