5kr1p7-k1dd13-RS4

Category: Cryptography

Help Arrow decipher the message Elliot sent to him!
flag format: isfcr{xxx}

Author: `Arrow#1334`

Flag: `isfcr{5kr1p7_k1dd13s_u53_RsaCtfTool_f0r_th3_w1n}`

This is a slightly more complicated version of the previous challenge (babyrsa). Do read the writeup for babyrsa to get a basic understanding about RSA before trying to understand the solution for this challenge.

The flag has been divided into 4 different parts, each of which is encrypted using different RSA implementations. Here, each RSA implementation contains a vulnerability that can be exploited to retrieve the plaintext message without having prior knowledge of the private key.

Flag 1

As can be see in the generation script, the two primes used are only 64 bits long. Factoring 64-bit integers can be done pretty fast. There are online lookup tables like this one which can be used to find factors of most small numbers very quickly. Once you find out the prime factors, following general decryption method will give you the first part of the flag.

Flag 2

Flag 2 uses exponent as `3` but two `1024` bit prime integers. One can conclude that the modulus `n` is a 2048/2049 bit prime integer. However, the flag is actually only 12 bytes (96 bits) long and even after encrypting using exponentiation to the power of 3, the number will be bounded by approximately a 300 bit value. As the modulus is too large, we can simply take the cube root of the ciphertext and obtain the plaintext.

Flag 3

The third modulus is generated by using nextprime() on an already generated prime. This often leaves a very small difference (~20 bits) between the two primes. Searching for attacks related to the same show Fermat attack and Shank's Square Forms attack. This is a good resource to learn more about it. After factoring the modulus, you can follow the normal decryption procedure and retrieve the third part of the flag.

Flag 4

We notice that the exponent is very large for the last part of the flag. Very large exponents may have very small inverse counterparts under modulo (i.e. the private exponent `d` might be bruteforceable). Searching for exploits related to the same, you can find and read more about the "Wiener attack". Using that, you can find the fourth part of the flag.

It's quite hectic to implement all the four attacks yourself by reading papers and research from others, and so there exists a tool that can be used to automate the attack process (for a few well-known attacks). You can find it here. This tool was hinted to using the long string of gibberish characters:


```
Vm0wd2QyUXlWa2hWV0doVVYwZG9jRlZ0TVZOWFZsbDNXa1JTVjFac2JETlhhMk0xVjBaS2MySkVUbGhoTVhCUVZteFZlRll5VGtsalJtaG9UV3N3ZUZadGNFdFRNVTVJVm10a1dHSkdjRTlaYlhSTFZsWmFkR05GZEZSTlZXdzFWa2QwYzJGV1NuUlZhemxhVmpOb2FGcFdXbUZrUjFaSVpFWlNUbFpVVmxsV1Z6QXhWREpHUjFOdVVsWmlSMmhvVm1wT2IyRkdXbGRYbGRYYlhSWFJWWmFlVmRyV25kV01ERldZMFZ3VjJKKVVJYZFpWRXBIVWpGT2RWWnNTbWxTTW1oWlYxZDRVMVl4WkdJHUllZbGhTV0ZSV1pGTmxiRmw1VFZSU1ZrMUVSa1pWYkZKRFZqSkZlVlJZZYUZkU1JYQklWbXBHVDJSV1VuUmpSazVYYVpOb1dsWXhXbXhbXROUjFGNVZXNU9XRmRIYUZsWmJGWmhZMVphZEdONlJrNVdiWFF6VjJ0U1UxWnJNVVZTYTFwWFlsaGG9NMVpxUm1GU2JVbDZXa1prYUdFeGNGbFhhMVpoVkRKT2MyTkZaR2hTTW5oVVdWUk9RMWRXV1hoYVJFSmFWbTE0VjFSVmFFHOWhiRXAwVld4c1dtSkhhRlJXTUZwVFZFZqRmtkVnBGTlZOaWEwcElWbXBLTkZReFVWZk1VlWUlZXbUZZsYkZweFVtMUdVZVMkpWVmpaWlZWcHdWRmxaWlUdORVdsZGlXRUpJVmtSVVpsSDJXbVEZWTkhvWFdVCRp4hHVWtVVFYWt0aU1XUlhWWlhHVWVVUtxEaJNRUdUJaVlZKR1F0SkRGSWZyhWRFRKUjdlTWZMRNFkZVkRVQ1QlF1U1NWbzBZVFUU1RMVV5SWnJSdVFVZllKYVFTFJ0
```


This is a Base64 encoded string. You can use an online decoder to decode the above string a few times (I think 12?) and retrieve the link to the RSA tool.

Here's the complete exploit script (Notice the `os.system()` calls in the comments. You can use the RsaCtfTool.py to automate the attack process):

```python
import os

import gmpy2
from Crypto.Util.number import long_to_bytes

# factordb attack
n1 = 1459337480598978327080196309026257134l3
e1 = 65537
c1 = 208158761132766196573115629933642352O6

# small exponent attack
n2 = 7994708512956649866296342059990363182858659414707015769987919460616339925764746891290734984728739752346143180951509437069026318349604016836744341820628817724571136076933864936269584222638337339193675965187068850776672136345353245190235901301029112183953296020869535095358575034855088917491978109960998449670
```

```python
e2 = 3
c2 = 257525149189366352852960879874042331066284428708350287695648784342424769083 13650494763

# fermat attack or Shank's square forms factorization attack
n3 = 113695613888837555189023112264154237220133996822923649393737806323929648088090684225351829943425730233924571317708190054744606421906075372472420115373440728793966647442907101863637418533060598187332117210579532708720029746230145723452869241108694376173570544798707987242808923153085562962232590668047363279461
e3 = 65537
c3 = 151105168306469854932137293952537091749651162038822074092752790243861403001533331169939293557958914027694336440254755496320760405553077670739862295701229468226893153800604144386656458370155953161553747069158734006177378121781699481931276018909367901158878036106393161856165363392348466024765172689541 59940092

# wiener attack
n4 = 338630205260455689413627911306068443537112802550361922213620660503310212139001530156458392949653034244789612680980241965923780722889133495349537107789761426092510299239678696031652780059016898519278860185536978111680123402473365833456785718098200501968322228116681190425490850863660038143310790555506293106653050174262471649179173093656763946257235681980586392230447218179278964626176124426615857733950102117938674282636936094069075258237416065546593509302494726576026227551920883962084579635168761189995794814926094510046419165007371450799003658587100556051088147493947712592469412133312536422828670173807709914587
e4 = 318540665379393469901456665807211509077755719995811520039095212139429238053864597311950397094944291616119321660193803737677538864969915331331528398734504661147661499115125056479426948683504604460936703005724827506058051215012025774714463561829608252938657297504427643593752676857551877096958959488289759878259498255905255543409142370769036479607835226542428818361327569095305960454592450213005148130508649794732855515489990191085723757628463901282599712670814223322126866814011761400443596552984309315434653984387419451894484613987942298157348306834118923950284809853541881602043240244910348705406353947587203832407
c4 = 231189770340753244576712062662952239691668632968587788927565229106586299124916852719002089108348788504148185674454906943170363923385219998624901134692599702458017079685282827689630291151023362900298038598041535860948072108402302189075174724723277724488575639731583342078239390952087064645434882965757571868590845317179753025874079372287503147979690225719527628968560015879291237059624400338835119607706789961742031115136617293985561417565790026423791206900566195850082475445909429778661223857291306148317308834196673091677299694057317044361556413668732199855317830253635794223003360353571416436091734 02740169428727

# you could either use RsaCtfTool (https://github.com/Ganapati/RsaCtfTool) or you could be cool and implement attacks yourself :)

# os.system(f'rsatool -n {n1} -e {e1} --uncipher {c1} --attack factordb')
# os.system(f'rsatool -n {n2} -e {e2} --uncipher {c2} --attack cube_root')
# os.system(f'rsatool -n {n3} -e {e3} --uncipher {c3} --attack fermat')
```

```python
# os.system(f'rsatool -n {n4} -e {e4} --uncipher {c4} --attack wiener')

def attack_1 ():
  # factors from factor db
  p = 10097308971066607687
  q = 14452736712134345299
  assert(p * q == n1)

  phi = (p - 1) * (q - 1)
  d = pow(e1, -1, phi)
  m = pow(c1, d, n1)

  return long_to_bytes(m).decode()

def attack_2 ():
  # using python exponentiation ** operator to get the cube root may not work due to precision errors
  # either use a good math library or code binary search

  # m = gmpy2.iroot(c2, 3)[0]

  low = 0
  high = 2 ** 1000

  while low < high:
    mid = (low + high) // 2
    product = mid ** 3

    if product < c2:
      low = mid + 1
    else:
      high = mid

  m = low
  return long_to_bytes(m).decode()

def attack_3 ():
  a = gmpy2.isqrt(n3)
  b = a * a - n3

  while True:
    if b < 0:
      a += 1
      b = a * a - n3
      continue

    s = gmpy2.isqrt(b)

    if s * s == b:
      break

    a += 1
    b = a * a - n3

  p = a - s
  q = n3 // p

  assert(p * q == n3)
```

```python

  phi = (p - 1) * (q - 1)
  d = pow(e3, -1, phi)
  m = pow(c3, d, n3)

  return long_to_bytes(m).decode()

def attack_4 ():
  def rational_to_continued_fractions (e, n):
    # convert e / n fraction into continued fraction partial quotients
    # https://en.wikipedia.org/wiki/Continued_fraction
    k = e // n
    quotients = [k]

    while k * n != e:
      e, n = n, e - k * n
      k = e // n
      quotients.append(k)

    return quotients

  def continued_fractions_to_rational (quotients):
    # converts a list of continued fractions x / y rational form
    if len(quotients) == 0:
      return 0, 1

    quotients = quotients[::-1]

    num = quotients[0]
    den = 1

    for q in quotients[1:]:
      num, den = q * num + den, num

    return num, den

  def continued_fractions_to_convergents (quotients):
    # converts continued fractions to convergents
    convergents = []
    slice = []

    for q in quotients:
      convergents.append(continued_fractions_to_rational(slice))
      slice.append(q)

    return convergents

  f = rational_to_continued_fractions(e4, n4)
  convergents = continued_fractions_to_convergents(f)

  for (k, d) in convergents:
    x = e4 * d - 1

    if k > 0 and x % k == 0:
      phi = x // k
      s = n4 - phi + 1
      D = s * s - 4 * n4

```

```
      if D < 0:
        continue

      sq = gmpy2.isqrt(D)

      if sq * sq == D:
        break

  m = pow(c4, d, n4)
  return long_to_bytes(m).decode()

flag1 = attack_1()
flag2 = attack_2()
flag3 = attack_3()
flag4 = attack_4()
flag = flag1 + flag2 + flag3 + flag4

print(flag)
```