PEM

September 15, 2023

1 PEM key format

We can generate key in PEM format using e.g. OpenSSL:

openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt rsa_keygen_bits:4096 Such generated key would be called RSA-4096, where 4096 is number of bits in the modulus.

```
[2]: from cryptography.hazmat.primitives import serialization
     with open('private_key.pem', 'rb') as key:
         private_key = serialization.load_pem_private_key(
             key.read(),
             password=None
     # private numbers
     d = Integer(private_key.private_numbers().d)
     p = Integer(private_key.private_numbers().p)
     q = Integer(private_key.private_numbers().q)
     # public numbers
     e = Integer(private_key.private_numbers().public_numbers.e)
     n = Integer(private_key.private_numbers().public_numbers.n)
     print(f''d = \{d\} \neq \{q\} \neq \{p\} = \{p\} \neq \{p\} )
     print(f"e = \{e\}")
     print(f"p has {p.ndigits()} digits ({p.nbits()} bits), q has {q.ndigits()}_u
      digits ({q.nbits()} bits), n has {n.ndigits()} digits ({n.nbits()} bits)")
```

 $\begin{array}{ll} d = 3374373417417346944917263625055478737643055156438276071850665758380053647446\\ 97882803784218778084478977846005006716635279247490723297691397961784754290476998\\ 18240365451954780224393784752548731101936704387437251886591324401958257501604069\\ 93842106052666936809173755998277097122964932830727621455065456849698228973650691\\ 81468490059185740275877104025199985233405538445728229840709439990858210888669657\\ 66528813799032742690596535512043808712927065935996348857362038031399314099061782\\ 71194412225092630691790706884145523124144194464034540393414973065723237211639026\\ 62838622254793361675022803315954572094608574607716741074528427448318513166309535\\ 63014224473989697441410633087795029450858932808164786753278192841824819662993461\\ 87181973695763889892817719311864682151136733097936843568983907296941036073120055\\ 31661755342253865223430310664021821399870959244859129223158343511685606619862463\\ 84480490956006494224478918593869425838517494977111943492173827477918134215940025\\ \end{array}$

 $17434511188159042447826303035989072792538382479381420406376711462423408991301891\\98647983185647413470138558806288452703193986579354971300948072166856865719162473\\50286595114419286861547523516507126277374395081163895631187974488317443798050269\\8356799151228115448677963245386554265$

 $\begin{array}{l} q = 3174030854376973432317343624720726990178005441884614263142527702682862071911\\ 83181291688672535277607838759740403269515381200842649536961077810418277687998694\\ 27266848765844034505962695267199977678381757241025662918638510855654917433203562\\ 91532823854181951240923095127676789850710894471525048133345379822389997333141204\\ 03286898313676401197714014913470583047163811663805027715089122357445990107706445\\ 18139804692210448064026957725152028116211296287736485645733389434720360968274049\\ 81029530343891023856660397069597532748664671566265988609848705346103216503890930\\ 5758823228750887731221834990850211094424016493970401176784493 \end{array}$

 $\begin{array}{lll} p = 2770213855637207509947874707238483454331438167790381990394288297638115163682\\ 53012561609981549198519447889895202190987343971213954944984739228851053245779130\\ 72494420447643526959987949354776121844043485552485906040664466646690733602460906\\ 24275484836150438438308743736091023078759804424428411518669438675454857445999277\\ 94669202874137665257076904578894643410900993115824867604618080470098917861398402\\ 25270389759971551962605361286394088451619376768215023911220871458931153992464901\\ 38925536185711072572207606106592981872552762601480030843212732729888668275817645\\ 1128911262128346044979815263481147458136854977937413336503663 \end{array}$

n = p*q = 879274425101509549238768662062188024447985788984514726734034757293759993211930545318739069860415987387026107515613221195818965430112552230456899822748 99870340976519144720124864226383304820143622189923785045962350227607994945511789 87110069550641882863334962740549364488296647729238304219779482455452658276163855 87797257145035454351419191919584148451941925345399946957658136907917964193766174 19095963252733348891284285902941736338659211379313089801363042558547982896015865 77842966715836158093720168040923470279555221796458986990670870224376340483835144 96512639706494276530792127875825755111201123728820107810866573516982908016033155 34673044555562468567255486143414927302750708494095754774469339919471395128252358 2897421484483081109795262152504101456097859

e = 65537

p has 617 digits (2048 bits), q has 617 digits (2048 bits), n has 1233 digits (4096 bits)

2 Encryption and decryption

Using RSA equation of form $(m^e)^d \equiv m \mod n$ lets encrypt a value of m = 22.

```
[3]: m = 22 m
```

[3]: 22

```
[4]: temp = m ** e temp
```

```
[6]: #temp.ndigits()
temp.nbits()
```

[6]: 292258

```
[7]: encrypted_m = (m ** e) % n encrypted_m
```

 $[7]: 28394959899460324726516362460156987013263620843358739545951608239527063471181284 \\ 20614146100842427404552705816774630998647691852547810146226096591328232455545843 \\ 70289109545842617547793211337488596455860434031342810143078040496490564567240050 \\ 76455796781256840270130243186243658242115806079554508571816667872850018547836220 \\ 43751967880193415730902152095824933514300572832451203820151324472470202751957125 \\ 25690106981610720754173147889504051796865309746521784588629918746387474605893442 \\ 44305126656940007921074969525028914632760113531257372549900006151269697739933086 \\ 57421274631871828346012245463425826732969590041787802122924658167030144066926523 \\ 17050232926705911901397833816108738227774856056086335085841656826760137322608294 \\ 20078111209245799707032688043715257973855360861844557093222595481567452855465248 \\ 42669752047347956025095512069150498101643456490272992251605263176819250390362840 \\ 57106792369234163316124889899066492107437361687218156110223076940525397359612406 \\ 89670797113053915699352458862132584398513162337784901717089154195731958729242279 \\ 20933259212424893804873643379281682448622306585481307982691494053799833055511928 \\$

45857961370890505218227362974553098372142642837899897398238600641650995557520208 772938644902328876357641104119375

We can now send public-key encrypted message over insecure channel to owner of corresponding private key for decryption.

Let's see how decryption works.

```
[0]: # encrypted_m ** d # won't work due to exponent too large for naive power
```

```
[8]: decrypted_m = power_mod(encrypted_m, d, n)
decrypted_m
```

[8]: 22

3 Signing and verification

... but wait, there's more.

Asymmetric cryptography can be used for signing and verification, where holder of private key signs a message which authenticity can be later verified using corresponding public key.

Let's create a signature s of the same message m=22 using our private key:

```
s \equiv m^d \mod n
```

```
[0]: # m ** d # won't work due to exponent too large for naive power
```

```
[10]: s = power_mod(m, d, n) s.nbits()
```

[10]: 4094

And now verify signature:

```
(m^d)^e \mod n \equiv s^e \mod n
```

```
[11]: temp_s = s ** e # might not work due to running out of memory
temp_s.ndigits()
```

[11]: 80755629

```
[12]: power_mod(s, e, n)
```

[12]: 22

Reminder to switch sharing to other tab.