1.

a = 4841247740021026788214420074996258540545281

b = 712010411572858151605922429225626518528001

n = a \* b = 344701879589854066393335389185933307811699084203169955034932

9339556642079028006913281

34470187958985406639333538918593330781169852887

73547956464389519214137857142947840000

The smallest valid RSA public key should be relatively prime to , so I write a python procedure to compute this value [1]. The principle is that starting from 2, add 1 each time until the greatest common divisor of it and equals 1 (relatively prime). Inputting the parameter , we get the smallest valid RSA public key e 179.

I write a python procedure to compute the multiplicative inverse using extended Euclid algorithm [2].

The inverse d of 179 mod is:

385141764904864878651771384565288612080110088131122676699931789856328252194742775419

So, the public key is

{179 (public key),

3447018795898540663933353891859333078116990842031699550349329339556642079028006913281 (n)},

The private key is

{385141764904864878651771384565288612080110088131122676699931789856328252194742775419 (private key),

344701879589854066393335389185933307811699084203169955034932933955664079028006913281 (n)}

2.

https://asecuritysite.com/encryption/random3

This webpage is an online prime number generator, I entered 1024 which means it will generate prime number of approximately 21024 (>10256).

a =

142067892537710038404728203368355245008715878078991689417267746803540137968406393230902896111023520403849882954090477365027830148120971559436319701479923864343284491573463263596198109898190549888399454023923106284617885316115699288094095879476855486606272662430203820611713137791734597634776843216536256907593

b =

39723044180851877063845739777264642760380297960701224406436285337819145031539713092931984772436287295410479811716201261331883217745105395871300583469588879022400357650863784681161122147080818767180261074244886377235388032702811104536171251064168207996618117479688549109949612934116332186286576428448828828279

n = a \* b =

5643369171955972550742291655346337369828313868176659207049306724895849492235000938973263052427377353912320911436213525896819039221666981632209771239550443541518564138548119142789995772155085324407980751295384682237712367896381151957937853698793987452967269976080051490782381123192328309953967796750479119434137174848700743264270013280889405885321856003698555642046141751883371500944319785655255954111369662702308195069903285548306489916669565676596431816073830633329205082569391920188985281933948062598563923073549788906782975140820819902641902555473287002521663288093693089719399015653300047886785682559486168222447

5643369171955972550742291655346337369828313868176659207049306724895849492235000938973263052427377353912320911436213525896819039221666981632209771239550443541518564138548119142789995772155085324407980751295384682237712367896381151957937853698793987452967269976080051490782381123192328309953967796750479119433955383911982181348801439337743785997552759827658862728222437719742012217944373679331421073227909855003047832304096606921946776550803488721288811531124317889963520233345064871911626049888676693942984207975381796244929701792002309510011635424932263307918772508183800719997736264927449118065722262914501082486576

The smallest valid RSA public key should be relatively prime to . By implementing SmallestPublicKey.py [1] we get the smallest valid RSA public key 5.

By implementing ExtendEuclid.py [2] we get the inverse d of 5 modis:

4514695337564778040593833324277069895862651094541327365639445379916679593788000751178610441941901883129856729148970820717455231377333585305767816991640354833214851310838495314231996617724068259526384601036307745790169894317104921566350282959035189962373815980864041192625904898553862647963174237400383295547164307129585745079041151470195028798042207862127090182577950175793609774355498943465136858582327884002438265843277285537557421240642790977031049224899454311970816186676051897529300839910941355154387366380305436995943761433601847608009308339945810646335018006547040575998189011941959294452577810331600865989261

<https://www.random.org/integers/>

This webpage is an online random number generator, I used it to generate a random message m (>10500) and a secret random number r between 0 and n-1:

m =

989341685881536206614879811226442176696794054183020113469687580534700488037777246497088795021641632603311279612984798417044537830394798241345256954980879751731525312571158114896194426315035791665639180362095298518921819575423824559093693272525402946572747524569730265327077859600082056160131745427634112489135044693876086916612762856183039921824068265551615711386910110799037683036465436615934031928287349232508877540167468968229931495901393252735488402915395226252851600289713026918253466980403281610701173519553982

secret random number r between 0 and n-1:

11510992366382457207158076992709663371704298857235825058760911018821333022530925576207612023676045043220428759862698334230089604565314608705154170271379279

The blinded message

199944811124900359263285134766585484562033380639515997686080578562160729014525151983686637056284276474024010506093430986818682275005205935609832477251611104790445831018613618113513856053380158617782068561671530217667968935153016577614854159418761177269126101211189598117054666596588275663520582308772326712654959098261722279072047920491109250770813738037838371902445243649265858372813678538719494962414555774752664974806984677771035217287794022234463343161980540742084585253221974218603413248340964463845897435693031797508321071905780033559591944907468943615992939480236065142961852046018524719659747188240146413574731500198441071204529983324965244686981742027547594807919965768860401884107399382084228304306751937339082633736624300074948333077659581740736055049543643703200004391464586109337617799353446752183118312123210014564960406374511669118081422544763048792109254297209002885453828730591191119919369869051669113070986066705786308188775836403372827574388172882455807698578254050300843885643857792807642107889711749408991859481466959348824675233911212114971081366691522651543475689376362235222969098174203084954031394442678566220509675230950066949557129557311996425656011485726008861223801054253194914523877895806356493488361080779784608737515231773359832097265807998823362115862818

By implementing Mod.py [3] we get the ciphertext sb of mb encrypted by secret key d:

1904355323618778421509433650983970574299644816471371382582674645240622109746871252856068622999796749001792761368556660699731622921131634838532729124797784120407053849518325563126613886769722229044026665491595061047506377517883039702322043690081931124268638282411315589811682062532749974316193780510174130640845329632759361210723865749056257818468138906518803876759400447525723254822002195990633149319351292584221491167510019480177976400697820987414640656630821017088560843021565437482231993378066083187187053339283708963601877746553551963422041118661106290108875437345872115088940607640564185034958814595424167440239

By implementing Mod.py [3] we get the correct ciphertext s of plaintext m encrypted by secret key d:

839496556837972706618248300847050209249860360358411006161578387531659397165110455091898594785301524687222479276748607246985141813311800977046584724244874228025521296858884399274048829205656040618389208486293809318243148726499209262398106412849839025738169910106426940030671582801707825659221835014225685081525020305586521598663303047450242710464852980734201020508838786671024347968144126026615298744855125724156148676710429792474523479628691888729668616724903176346982821887852343316861139680124558680684343807380248734826676095787407612951191170389807131053746886197844156192733154975352869188044353974941020603122

By implementing ExtendEuclid.py [2] we get the inverse r-1 of r mod nis:

2969204593263916667513352194730924843641692120889681054689502948828543712768903801604589110809721392566964511479496964787887351848520189708579193602738456884269110223020769138994417304970344105687671916937306225690129824367561461181777769742121109301581535980149842890356373946167518913848820642471059047784584745228173302436074844559220537391985670421605302728041232218592307971988614855945569545736635669615314455530595915770478851554335827175457302487146926826415109706267097421944776110203008697721792172681566670519441774105626988234714428994593618510347142909668726784571382474305683468124074969271819864145343

The ciphertext sb of blinded message mb multiply by the inverse r-1 of r mod n and the ciphertext s of plaintext m should be congruent modulo n. By implementing Mod.py [3] we get:

839496556837972706618248300847050209249860360358411006161578387531659397165110455091898594785301524687222479276748607246985141813311800977046584724244874228025521296858884399274048829205656040618389208486293809318243148726499209262398106412849839025738169910106426940030671582801707825659221835014225685081525020305586521598663303047450242710464852980734201020508838786671024347968144126026615298744855125724156148676710429792474523479628691888729668616724903176346982821887852343316861139680124558680684343807380248734826676095787407612951191170389807131053746886197844156192733154975352869188044353974941020603122 = s

In this way, we can use the blinded message to recover the original plaintext.

3.

Man-in-the-middle attack:

Suppose A and B want to exchange keys, attacker can disguise himself as B in order to communicate with A. By doing so, the attacker shares a secret key k1 with A. After that, the attacker disguises himself as A to exchange data with B so that he gets another secret key k2 with B.

Attack success.

1. Message Authentication Codes

By applying MAC, the secret key for MAC is only shared by A and B which means for A (B), only B (A) can generate the correct MAC. The adversary does not know this secret key, so he cannot generate a correct MAC for a given message. As a result, when the attacker wants to disguise himself as A or B to send a wrong message, the receiver B or A will find that the message has been modified through MAC authentication.

Attack failed.

1. Public Key Digital Signatures

Digital signature method is more secure than MAC, because only person with a private key can generate the correct signature. If the adversary wants to disguise himself as B, he cannot generate the correct signature of B, so the receiver A can realize that the message has been modified or is totally wrong.

Attack failed.

1. Hash functions.

There is no secret key shared by A and B, so the adversary can also successfully disguise by generating the correct hash value. The attack method is the same as above.

Attack success.

4.

This method is really easy to break. Because each alphabet character is encrypted separately, there are only 26 different cipher characters in ciphertext corresponding to letter a to letter z. The most efficient attack against this method is chosen ciphertext attack (CCA). Different from the attack method in textbook, we can simply choose these 26 different cipher characters and get the plaintext. The cipher characters in the ciphertext and the alphabet characters in the plaintext are one-to-one correspondence.

If you cannot use this attack method, cryptanalytic attack is also practical. Because ciphertext reflects the frequency data of the original alphabet. Without confusion and diffusion, attacker can easily break this encryption system by using statistical characteristics.

A simple countermeasure to these attack above is to encrypt the plaintext in blocks. Notice that the binary value of each block should be less than n. In addition, you can also add secure hash function to enhance security (integrity).

5.

(a)

Step 1: A sends a message to responder B. The message includes identity of A (IDa) and a unique identifier Na encrypted by Ka. Na is a nonce generated by A and Ka is the master key known only to KDC and A.

Step 2: B sends a message to KDC which can be divided into two parts. One is the message just received from A, the other includes identity of B (IDb) and a unique identifier Nb encrypted by Kb. Nb is a nonce generated by B and Kb is the master key known only to KDC and B.

Step 3: The KDC responds a message which can be divided into two prats as well. One is for B encrypted by Kb and it contains three items which are the session key Ks, identity of A, nonce Nb. The other part is for A encrypted by Ka which contains the session key Ks, identity IDb and nonce Na.

Step 4: B stores the session key Ks for future use and sends a message originated by KDC for A.

(b)

Consider the most extreme case, there are n entities want to communicate in pairs, so there are n(n-1)/2 session keys needed simultaneously. Let’s assume n equals 10000. For a certain entity, it should store its master key shared with KDC and 9999 session keys shared with other entities for each connection. For KDC, if it doesn’t need to store the session key history, it should distribute 50 million session keys for each connection and only needs to store 10000 master keys for these 10000 entities.

(c)

As we can see clearly, the alternative key distribution method has 4 steps and the scheme in the lecture (textbook) has 5 steps, so the efficiency of alternative key distribution method is better (4 < 5). The fourth and fifth steps of the scheme in the lecture (textbook) is added for authentication. By sending nonce N2 encrypted by the session key Ks to the initiator A, only A can respond a message with f(N2) encrypted by Ks. So the scheme in the lecture (textbook) provides authentication but the alternative method does not. In the alternative method, two nonce Na and Nb are used so that they can assure the message is not a replay. As for the method in lecture (textbook), two nonce N1 and N2 are used to assure the message is not a replay. So both of the methods can resist replay attack.

The session key in both of the methods is encrypted by Ka and Kb all the time, so the confidentiality is assured. Due to alternative method cannot provide authentication, I think the method in the lecture (textbook) is more secure than the alternative method.

6.

This hash function is weak, so it’s very easy to find another message which has the same hash value with M.

1. One-way property

For a very large n, the hash function has one-way property. Even though n is not a large number whose factorization is unknown, it’s also really hard to find the plaintext. For example, let’s assume n is 10 and the hash value is 7. It’s almost impossible to find the original message on the basis of these known information.

1. Second preimage resistance

Second preimage resistance means that given a message, it’s computationally infeasible to find another message with the same hash value. In this case, this hash function doesn’t satisfy the requirement. Let’s assume the hash value h, you can simply find another message whose quadratic sum is h or h + xn (x-positive integer). For instance, h is 123456789 and n is very large. You can always easily find a message which = 123456789 or 123456789 + xn (x-positive integer).

1. Collision resistance

Because this hash function doesn’t satisfy second preimage resistance, it also doesn’t satisfy collision resistance. In this case, you can choose whatever message you want and easily find collisions. For example, a message {4, 2} which quadratic sum is 20. It must have the same hash value with {4,1,1,1,1}, {3,3,1,1}, etc.

Appendix

1. SmallestPublicKey.py

def gcd(int1, int2):

if int1 % int2 != 0:

return gcd(int2, int1 % int2)

return int2

def smallestPublicKey(integer):

a = 2

while gcd(integer, a) != 1:

a += 1

return a

1. ExtendedEuclid.py

def extendedEuclid(a, b):

remainder = -1

x1 = 1

x2 = 0

y1 = 0

y2 = 1

if a > b:

divisor = b

else:

dividend = b

divisor = a

while remainder != 0:

gcd = remainder

quotient = dividend // divisor

remainder = dividend % divisor

temp1 = x1 - x2 \* quotient

x1 = x2

x2 = temp1

temp2 = y1 - y2 \* quotient

y1 = y2

y2 = temp2

dividend = divisor

divisor = remainder

if a > b:

return [x1,y1,gcd]

else:

return [y1,x1,gcd]

1. Mod.py

def mod(m, d, n):

i = 0

ex = 1

r = m % n

dictionary = {i: [r, ex]}

while ex < d:

i += 1

ex \*= 2

r = (r \*\* 2) % n

dictionary[i] = [r, ex]

result = 1

while d > 0:

if d - dictionary[i][1] >= 0:

d = d - dictionary[i][1]

result = (result \* dictionary[i][0]) % n

i = i - 1

return result