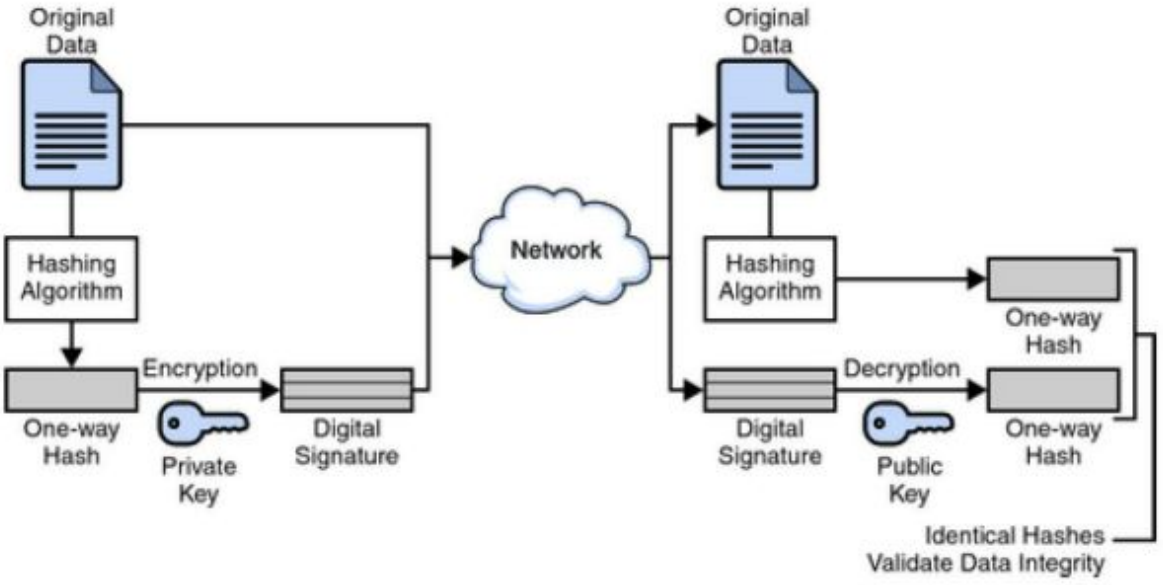
DAT 510: Assignment 3
Signature Scheme
Author: Sondre Tennø

Abstract
I solved this assignment by implementing the several building blocks that make up a Signature Scheme. I chose my digital signature based on the RSA encryption, and sha256 for my hash. I created a client-server model to exchange messages between Alice and Bob successfully and verifying the signature.

Introduction
In this assignment the goal was to implement a secure communication scenario, with a signature scheme verification. The program is composed of the building blocks that make up a signature scheme. The program will use hashing, encryption/decryption and digital signature. Implementation of the program is based of the illustration below.



Design and Implementation
To tackle this problem, I have made a very simple client-server model to represent Alice and Bob. In Python I have programmed a client and a server that will send messages to each other based on the Signature Scheme and verify its messages.

When I start the server, the server will use RSA key generation to create variables ‘e,n,d’. Where ‘n’ is the modulus for the public key and private keys, ‘e’ is released as the public key exponent and ‘d’ is kept as the private key exponent. This is based on the steps from the RSA wikipedia page.

1. Choose two different large random prime numbers p and q
2. Calculate $n = pq$
   - n is the modulus for the public key and the private keys
3. Calculate the totient: $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$, and e is co-prime to $\phi(n)$ **i.e.:** e and $\phi(n)$ share no factors other than 1; gcd(e, $\phi(n)$) = 1.
   - e is released as the public key exponent
5. Compute d to satisfy the congruence relation $de \equiv 1 \pmod{\phi(n)}$ **i.e.:** $de = 1 + x\phi(n)$ for some integer x. (Simply to say : Calculate $d = (1 + x\phi(n))/e$ to be an integer)
   - d is kept as the private key exponent

Snippet of the wikipedia page explaining the step process.

```python
def RSA_keygen(n=512):
    """
    Perform steps 1. to 5. in the RSA Key Generation process.
    """
    # step 1
    import generate_primes
    p = generate_primes.generate_primes(n=n, k=1)[0]
    q = generate_primes.generate_primes(n=n, k=1)[0]
    # step 2
    n = p * q
    # step 3
    phi_n = (p - 1) * (q - 1)
    # step 4 and step 5
    while True:
        e = random.randrange(1, phi_n-1)
        if math.gcd(e, phi_n) == 1:
            # step 5
            gcd, s, t = eea.EEA(phi_n, e)
            if gcd == (s*phi_n + t*e):
                d = t % phi_n
                break
    return (e, n, d)
```

Snippet of my RSA keygen code, going through the steps from the wikipedia page steps.

```
------------------------------
public Key of server: (e,n) = (2991445969926887687905479871927238186082366038
Private Key (d) = 2525880753715614014068953417067892454222533266342933097601
Public key of server save to pubser.txt file successfully
------------------------------
```

Snippet of the logs when I start the server.

‘e’ variable of server:

29914459699268876879054798719272381860823660388977633985963426328174143855047770980946143379600735737305418890768633821560463596690111423677454478412750959761536652097706022786047426232108347579861304 6

610895056543397688884569587610790899002382283053947763491148237664403606388418347930430557883850841708665 83

‘n’ variable of server:
48794259472344631035227827707045702244023051526847815498699762048235758005585758823612295232004928313553970714040213078298812824641351044521915109070591658272607451987039925755623346371061812359546044107243727951934760577686328616286620904649055688314039243594324813640232639597203993062406161630437524346137

‘d’ variable of server:
25258807537156140140689534170678924542225332663429330976012482473804953099254877163449685152916031254832510164151414748046373547836058009815854537868756843890124414750219306611049369229265743212547756273468166514226665983500608970764813556490575143530990945627542148591640951146826152407170140358948248714247

I then save the public key component ‘d’ to pubser.txt file so I can retrieve it on the client later.

The server is now running and is open for connection from a client.

I then start the client, which will also make the same variables as the server did. The ‘e’, ‘n’, ‘d’ variables. Where ‘n’ is the modulus for the public key and private keys, ‘e’ is released as the public key exponent and ‘d’ is kept as the private key exponent.

```
------------------------------
Public Key of client (e, n) = (178265408706319469776147473169774338
Private Key (d) = 815836991100793777783929244913129666194562727834
Public key of client save to pubcli.txt file successfully
------------------------------
```

Snippet of logs when I start the client.

‘e’ variable of client:
17826540870631946977614747316977433867657679618216651234745163447681616146349659157350848249692935957135573659199320394047548392257369934051413238164439163468304530356903301004396331407957511131046277495954621770103819904652434223204638260104606370657454267868520960346166397268541474237918222205267869262 09

‘n’ variable of client:

22854619740849179989565413784302706106855263471787916837225316330406223181471883012000056902609722499646756876533961982467796307795826426570786116421318400576027533476190273792319120248331352459410621344889547923025860136996522940512534656302537537125975220624435202090204763079264761101978444698435891756177

‘d’ variable of client:

81583699110079377778392924491312966619456272783457260294187794502701058098848112554744364554365900588693383982867318672131024620164457127848445499865490515228429220215729963721320697649831420636791085446938204195235581552817862301574029973965425967885388193699465037755526368991806280180205859744208720530 89

The server and the client will now connect and receive each other's public key and modulus variable.

```
------------------------------
Connection from: ('192.168.10.168', 3882)
public key of client is saved in recpubcli.txt file
received public key of client : 178265408706319469776147473169774
------------------------------
```

Snippet from server when connecting to client.

```
------------------------------
public key of server is saved in recvserpub.txt file
public modulus of server is saved in recvserpub.txt file
------------------------------
```

Snippet from client when connecting to server.

We can see that the public key received from the client, on the server side, matches the public key generated by the client. We will now start sending messages between the client and the server.

```
print ("Enter your message")
text = input(" -> ")  # take input
x = hashFunction(text)
#print('Plaintext x={}'.format(x))
x = int(x, base=16)

#print("value to encrypted: ", x)
message = RSA_encrypt(x, (d, n))
```

Snippet from the code sending a message from the client to the server.

We send the message from the client to the server, and use sha256 hash function to hash the message.

On the server-side we receive the message. We hash the received message and compare it to the decrypted rsa. If they match, we can validate the data integrity.

```
print("------------------------------")
print("From Bob: ")
print("original msg: ", str(dataq))
x = hashFunction(dataq)
x = int(x, base=16)
print("Msg to encrypt: ", x)
data = RSA_decrypt(int(datar), (int(recvData.decode('utf-8')), int(recvDatan.decode('utf-8'))))
if data == x:
    print("Signature Value: " + str(data))
    print("signature verified :)")
```

Snippet from the server receiving a message code.

We will see signature verified in the logs. We can also see it in the code when we're doing it reversely, client sending message to the server.

```
print("Enter Your message:")
data2 = input(' -> ')
x = hashFunction(data2)

x = int(x, base=16)

data = RSA_encrypt(x, (d, n))
conn.send(str(data2).encode())  # send data to the client
conn.send(str(data).encode())  # send data to the client
```

Snippet of server sending message to client code.

```
datamsg = client_socket.recv(1024).decode()  # receive response
datarec = client_socket.recv(1024).decode()  # receive response
print("------------------------------")
print("From Alice: ")
print("original msg: ", str(datamsg))
x = hashFunction(datamsg)
x = int(x, base=16)
print("Msg to encrypt: ", x)
data = RSA_decrypt(int(datarec), (int(recvData1),int(recvDatan1)))
#print('From Alice: ' + str(data))  # show in terminal
if data == x:
    print("Signature Value: " + str(data))
    print("signature verified :)")
```

Snippet of client receiving message from server.

Test Results

To test the results, I will run the program several times with different messages and show all the output logs. We will see if the messages come out correctly, the messages get verified, that the program is stable, and if the received keys match.

I will do three tests, sending messages between the client and the server and study the logs.

Test 1:

Logs from starting the server:

```
------------------------------
public Key of server: (e,n) = (498102315314853145358960263141833
Private Key (d) = 206590551408564736597011497197333626712113125
Public key of server save to pubser.txt file successfully
------------------------------
```

‘e’ variable server:
49810231531485314535896026314183329639121366903396724167652523966087840297027650363635387944994376705700434537385492379528909152380031647108174265781986909269023316492440879487433998479497117973441531663075810587451347466479885149490616098880906022212575796311396112557808060228245659404500770321491865956 43

‘n’ variable server:
29665276474001929377343704853603164109711978260029418777818934517100792438790988538766940461296283419030624683005580431990945749181529394931984958432788947763537652148899652943259725961485633116550209 2

866588779576541534494423902327640748026846735567122307876314309265247141611417841106958525414339366154929 03

‘d’ variable server:
206590551408564736597011497197333626712113125036021371484441085453444021335222099596366135995237243251159847211293095343179055368637234826170574045140158623781382187402208137137452156635697112365184196722469387571170205466811006233436441067906046044988471241262317317808892891977566166344868138889165560075 87

Logs from starting the client:

```
------------------------------
Public Key of client (e, n) = (169720792408947976129460618604760848
Private Key (d) = 234600354944213194543567376390643015606724609692536
Public key of client save to pubcli.txt file successfully
------------------------------
```

‘e’ variable client:
169720792408947976129460618604760848626347070660372115724910050193579074237251108851536797576546546374315564597737786228341826276902047336700027887608725244374784344952897043481026021494767276424945291405826768264397487042850681890186747615645383427059156155432335854678442990204385620708286887666709713360 69

‘n’ variable client:
622911291890160897132633181911594978289454375304084356615388522794474121325860898293960396224008119539538354589385046420158568602011234126220021353505412595275687207825409811165064070376855916869618407384946562284120297388545262798530966330828382653084349174561814389694715936077081299214971527464237738763 63

‘d’ variable client:
234600354944213194543567376390643015606724609692536646013150674034640337560423846760050467541970692135283417383216158068627742896631081164948015775693355501587340908279471752160814997754447475152797441540146443768892104083943612927805719622054464406742544998550688731486100426230648454655249605054381155295 17

Logs from connection between client and server:

```
------------------------------
Connection from: ('192.168.10.168', 23453)
public key of client is saved in recpubcli.txt file
received public key of client : 169720792408947976129460618604760
------------------------------
```

Received public key from client:
169720792408947976129460618604760848626347070660372115724910050193579074237251108851536797576546546374315564597737786228341826276902047336700027887608725244374784344952897043481026021494767276424945291405826768264397487042850681890186747615645383427059156155432335854678442990204385620708286887666709713360 69

We can see that the public key generated by the client and the received public key on the server do match.

Logs from sending message from Client to Server.

```
------------------------------
Enter your message
 -> Hello Server, From Client
```

Message: Hello Server, From Client

Logs from receiving message on the Server from Client.

```
------------------------------
From Bob:
original msg:  Hello Server, From Client
Msg to encrypt:  178179270448953419264211316945721809035387704323323919342726696727475350574 39
Signature Value: 178179270448953419264211316945721809035387704323323919342726696727475350574 39
signature verified :)
------------------------------
Enter Your message:
 ->
```

Message to encrypt:
178179270448953419264211316945721809035387704323323919342726696727475350574 39

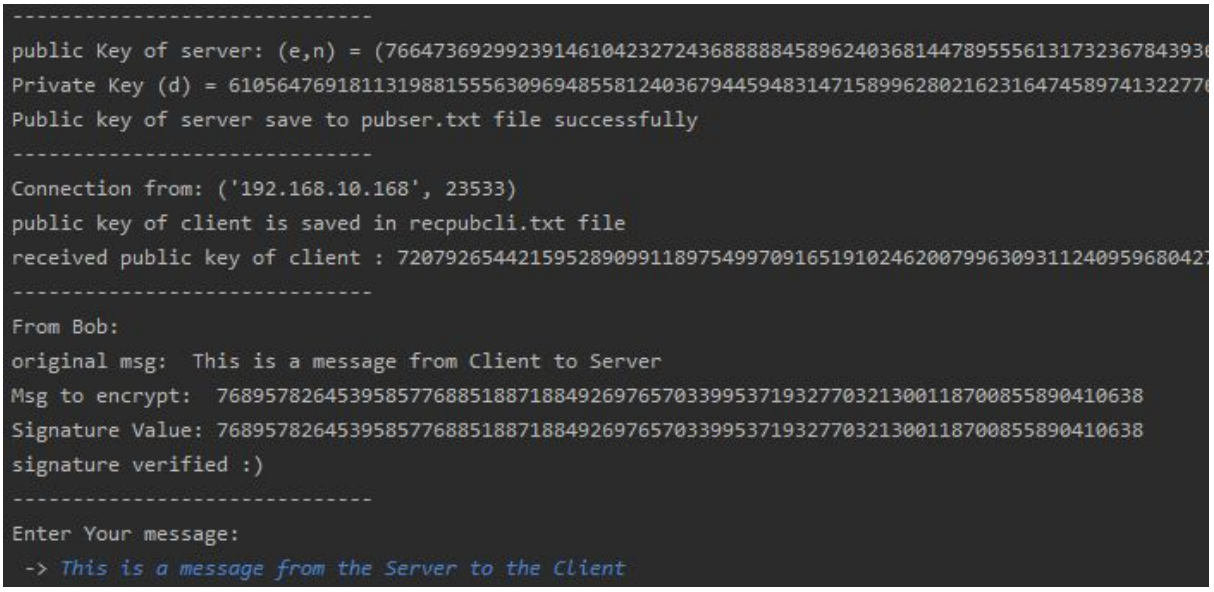
Signature value:
178179270448953419264211316945721809035387704323323919342726696727475350574 39

Results of Test1:

We can see that the hashed message, and the decrypted digital signature match, therefore we can validate the data integrity, and the signature is verified.
We can see that all the messages received are correct in plain text.
We can see that the public key received is the same as public key generated.

Test 2:

Logs from server:



‘e’ variable server:
766473692992391461042327243688888458962403681447895556131732367843936074591841428793924160176623428315587141516437503855617436145093074319073759079350282300205476910916717842728928500413054612169107940808284048946963401478105979055322558456875475885099801022177546937832800026612270609244837095976354825307

‘n’ variable server:
142065657105964884814513758507084885808581769320451893790301519599826568548900457934889851843932988542924905116198306359278469706813719823967261037581772297209828359830408461410204211115126506788358921053628924272514787474641349228788478956169169019657035538214988300753853886918036828063041356500323315723249

‘d’ variable server:
610564769181131988155563096948558124036794459483147158996280216231647458974132277605962415001618030915775902194947714912377618806266480456707203157256305659978541478608263965101158716857071284951549578584994657677831743962008972783701581864990911601639292319777989638128192107846418215064171955236538213368 3

Received public key from client:
720792654421595289099118975499709165191024620079963093112409596804275020878687336049816010399802464983362951105737749159050923833713233780830627329087833689300072870849946472490714407582852359380805037

5292826582340958718549626411546311578209246589015746185261304027396
4904943428086099386686101806985548143363

Message to encrypt:
7689578264539585776885188718849269765703399537193277032130011870085
5890410638

Signature value:
7689578264539585776885188718849269765703399537193277032130011870085
5890410638

Signature verified.

Received message:
This is a message from Client to Server

Sent message:
This is a message from the Server to the Client

Logs from client:

```
------------------------------
Public Key of client (e, n) = (720792654421595289099118975499709165191024620079963093112409596804275026
Private Key (d) = 9653888388073058411103894911278653941621042229300378145694365266740290129192436234703
Public key of client save to pubcli.txt file successfully
------------------------------
public key of server is saved in recvserpub.txt file
public modulus of server is saved in recvserpub.txt file
------------------------------
Enter your message
 -> This is a message from Client to Server
------------------------------
From Alice:
original msg:  This is a message from the Server to the Client
Msg to encrypt:  9537503812267631662703133081232956002249618243072522775218797976114799784461 6
Signature Value: 9537503812267631662703133081232956002249618243072522775218797976114799784461 6
signature verified :)
Enter your message
 ->
```

‘e’ variable client:
7207926544215952890991189754997091651910246200799630931124095968042
7502087868733604981601039980246498336295110573774915905092383371323
3780830627329087833689300072870849946472490714407582852359380805037
5292826582340958718549626411546311578209246589015746185261304027396
4904943428086099386686101806985548143363

‘n’ variable client:

985020197370529652185173512583155206700623135239071738456968527593166929330775217976883666685414808892847845717397300400168009487872299466427180604733693412902270368164690986595871143648862306451255508906447992729995381866833443590372584389728455300006774430885427303441235746280663709638207950101867740434 73

‘d’ variable client:
965388838807305841110389491127865394162104222930037814569436526674029012919243623470205341197425718773469710559647511140907582821278716309787485164868619639688091674273116265525553255130550176982809222965490048546057869334044390376979711137951328333401438510316220839733122209783856351652909166842766945316 67

Message to encrypt:
953750381226763166270313308123295600224961824307252277521879797611479978446 16

Signature value:
953750381226763166270313308123295600224961824307252277521879797611479978446 16

Signature verified.

Received message:
This is a message from the Server to the Client

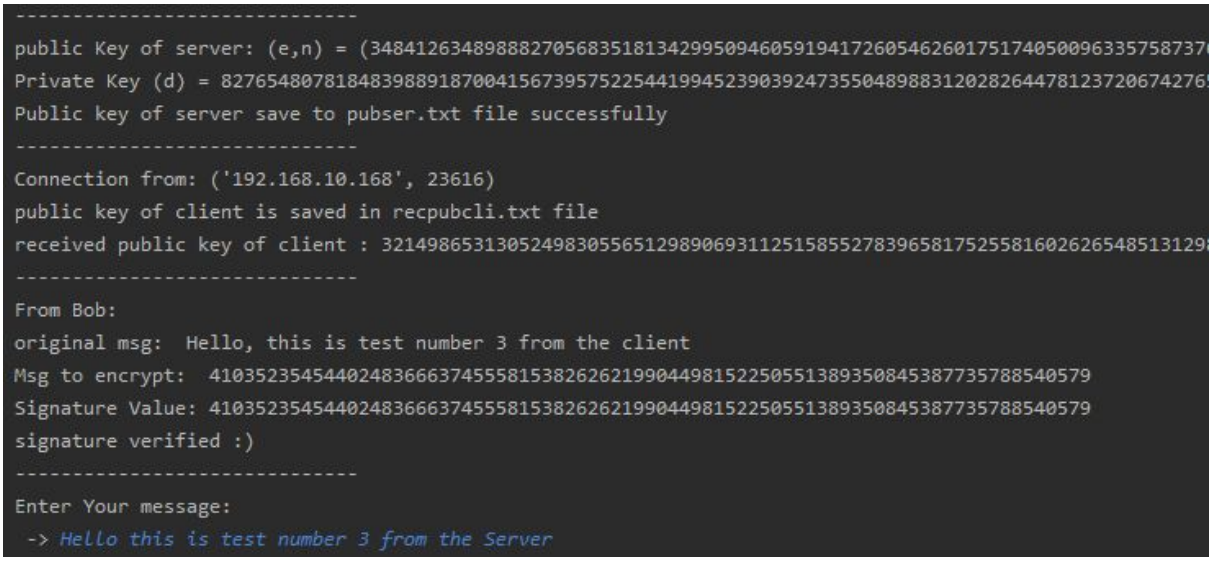
Sent message:
This is a message from Client to Server

Results of Test2:

We can see that the hashed message, and the decrypted digital signature match, therefore we can validate the data integrity, and the signature is verified.
We can see that all the messages received are correct in plain text.
We can see that the public key received is the same as public key generated.

Test 3:

Logs from Server:



‘e’ variable server:
348412634898882705683518134299509460591941726054626017517405009633575873765232355544741284833305576754343153985159839451330673118522266022083088459390075806000810287383600661008986977349578518871062368995380013302372657795142918837109137660002725744987417320046535480022059256646503999829384156394997739700 67

‘n’ variable server:
124614605530400410803380257597512175426079881571652585129774728934249915058538334955757346026108246894387518275856230617598637523126314110919969786908643233218494877995922653551535659120517600504491171904105585624162258352186013310826019534813271309872055990136195456126775236610540942575636635133963626779 99

‘d’ variable server:
827654807818483988918700415673957522544199452390392473550489883120282644781237206742765965528876196952692155454400443967156914405191457729182425640442549569329401155829602539817081379616201595244336779029214689429635575841168327919067176275497975312230790829086614705340443550671337124490182545230787003667 95

Received public key from client:
321498653130524983055651298906931125158552783965817525581602626548513129818766425620686635382114895096430905915218201756999501535159931433695934128320610676549223239672469530497302449037561277089652929391437886053482112098908322475523768850940540193511183527870743681453114817657433565896286035415718494939 3

Message to encrypt:

4103523545440248366637455581538262621990449815225055138935084538773
5788540579

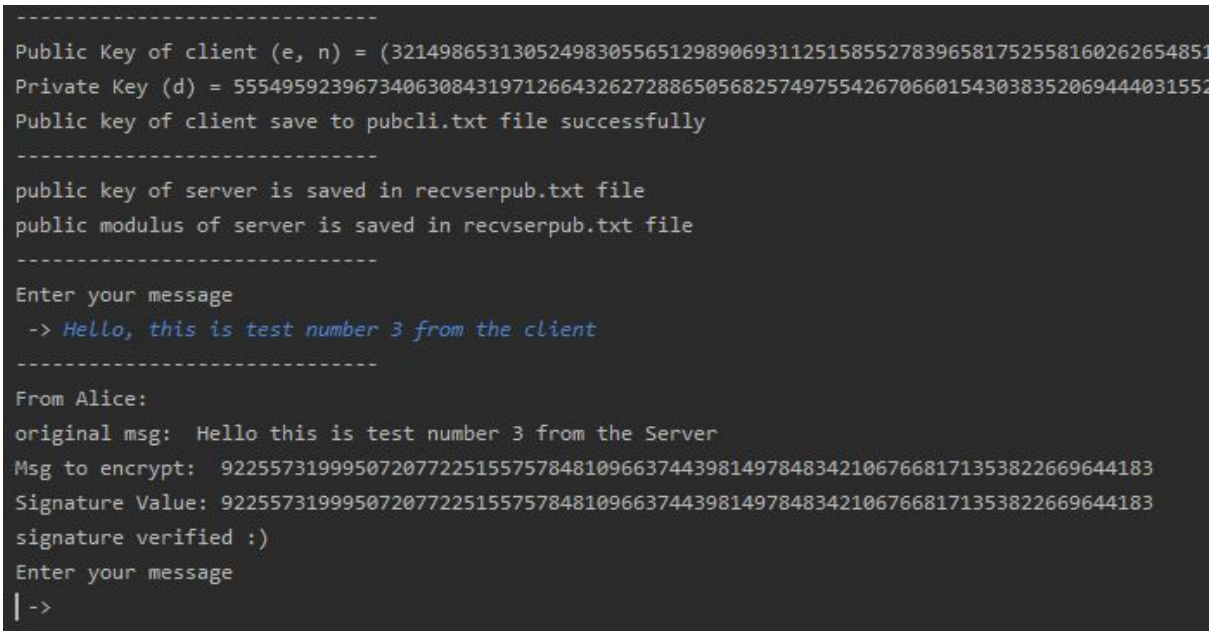
Signature value:
4103523545440248366637455581538262621990449815225055138935084538773
5788540579

Signature verified.

Received message:
Hello, this is test number 3 from the client

Sent message:
Hello this is test number 3 from the Server

Logs from Client:



‘e’ variable client:
3214986531305249830556512989069311251585527839658175255816026265485
1312981876642562068663538211489509643090591521820175699950153515993
1433695934128320610676549223239672469530497302449037561277089652929
3914378860534821120989083224755237688509405401935111835278707436814
5311481765743356589628603541571849493 93

‘n’ variable client:
8112718978383699156054826310898475219566311720650990834100200213144
5707613439442494754834518574583504174109074269178266909242945819796
5528693083581023250619526491554885245666429930534384767130054766927

3446033221307228772305304402738975864801959313386674258504315770931
6185139436589314731049123618253717236 43

‘d’ variable client:
5554959239673406308431971266432627288650568257497554267066015430383
5206944403155211225911834675160007137201228491218209798919212023488
5115453154829072747365520671550340211346416053579349188927562500314
5132296889847232770286300703955620450768249111102472224682803372556
8933390125564771512554743848483540675 37

Message to encrypt:
9225573199950720772251557578481096637443981497848342106766817135382
2669644183

Signature value:
9225573199950720772251557578481096637443981497848342106766817135382
2669644183

Signature verified.

Received message:
Hello this is test number 3 from the Server

Sent message:
Hello, this is test number 3 from the client

Results of Test3:

We can see that the hashed message, and the decrypted digital signature match, therefore we can validate the data integrity, and the signature is verified.
We can see that all the messages received are correct in plain text.
We can see that the public key received is the same as public key generated.

Discussion

In the test results we can see that all three tests gives us the correct message on the other side of the communication in plain text. All three tests had their message verified by seeing if the decrypted digital signature and the hashed message matched. And all three tests had the generated public key generated matched the received public key on the other side of the communication.

All three tests passed all our criterias, giving us a successful testing process.

Conclusion

In this assignment we learned about implementing a digital signature scheme. How a digital signature scheme is set up and the components of all its building blocks. In this assignment I learned about signature scheme DSS and hash functions, and how a digital signature scheme works. In the results we can see that all three tests gives us the correct message on the other side of the communication in plain text. All three tests sign and verify the message correctly. And all three tests receive the same public key as generated, meaning we get the correct public key.

I did use a finished library for generating primes(generate_primes.py), extended euclidean algorithm (eea.py) and for hasing I used sha256 library.

Works Cited

https://github.com/NikolaiT/Large-Primes-for-RSA/blob/master/generate_primes.py
Library I used for generating primes.

https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm
Library I used for the Extended Euclidean Algorithm.

https://en.wikipedia.org/wiki/RSA_(cryptosystem)
Information I used regarding RSA.

https://en.wikipedia.org/wiki/Digital_signature
Information I used regarding Digital Signature.

Stallings, William. Cryptography and Network Security: Principles and Practice (7th Edition). 2016

https://en.wikipedia.org/wiki/SHA-2
Information I used regarding SHA hashing.

Part 2 Questions:

1) What are the different types of SSL's and how different they are in aspect of security? Why ?

   There are three types of different SSL's. Oraganization validated, domain validated and extended validation.

2) Research about the the Certificate Authority Security concerns and explain

   Domain validation suffers from certain structural security limitations. In particular, it is always vulnerable to attacks that allow an adversary to observe the domain validation probes that CAs send. These can include attacks against the DNS, TCP, or BGP protocols (which lack the cryptographic protections of TLS/SSL), or the compromise of routers. Such attacks are possible either on the network near a CA, or near the victim domain itself.(Gathered from wikipedia: https://en.wikipedia.org/wiki/Certificate_authority#Validation_weaknesses)

   If the CA can be subverted, then the security of the entire system is lost, potentially subverting all the entities that trust the compromised CA.

   For example, suppose an attacker, Eve, manages to get a CA to issue to her a certificate that claims to represent Alice. That is, the certificate would publicly state that it represents Alice, and might include other information about Alice. Some of the information about Alice, such as her employer name, might be true, increasing the certificate's credibility. Eve, however, would have the all-important private key associated with the certificate. Eve could then use the certificate to send digitally signed email to Bob, tricking Bob into believing that the email was from Alice. Bob might even respond with encrypted email, believing that it could only be read by Alice, when Eve is actually able to decrypt it using the private key.(Gathered from wikipedia: https://en.wikipedia.org/wiki/Certificate_authority#CA_compromise)

3) How does browsers identify secure CA's From another CA's and how is it measured ?

   Your browser (and possibly your OS) ships with a list of trusted CAs. These pre-installed certificates serve as trust anchors to derive all further trust from. When visiting an HTTPS website, your browser verifies that the trust chain presented by the server during the TLS handshake ends at one of the locally trusted root certificates.(Gathered from

https://security.stackexchange.com/questions/158997/how-does-my-browser-inherently-trust-a-ca)

The longer the CA has been operational, the more browsers and devices will trust the certificates the CA issues. Basically SSL certificates encrypt the data that goes from your computer to the target website and back.
Browser connects to a server on a SSL port. Server sends back its public key. Your browser decides if it's ok to proceed, it checks if the public key isn't expired, and if it is verified or signed.(Gathered from https://www.quora.com/How-browser-come-to-know-that-SSL-certificate-provided-by-server-is-signed-by-CA)