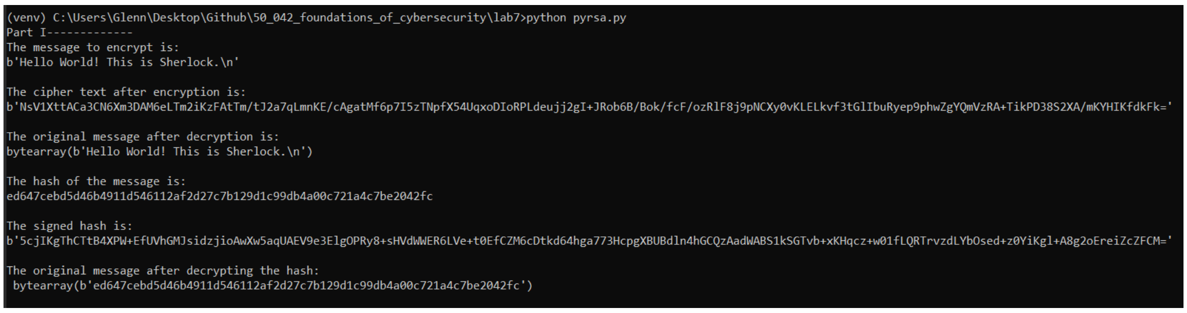
1. Installations

4. Install `pycryptodome` since the documentation shows that it supports newer python versions https://pypi.org/project/pycryptodome/

```
pip install pycryptodome
```

2. Part I

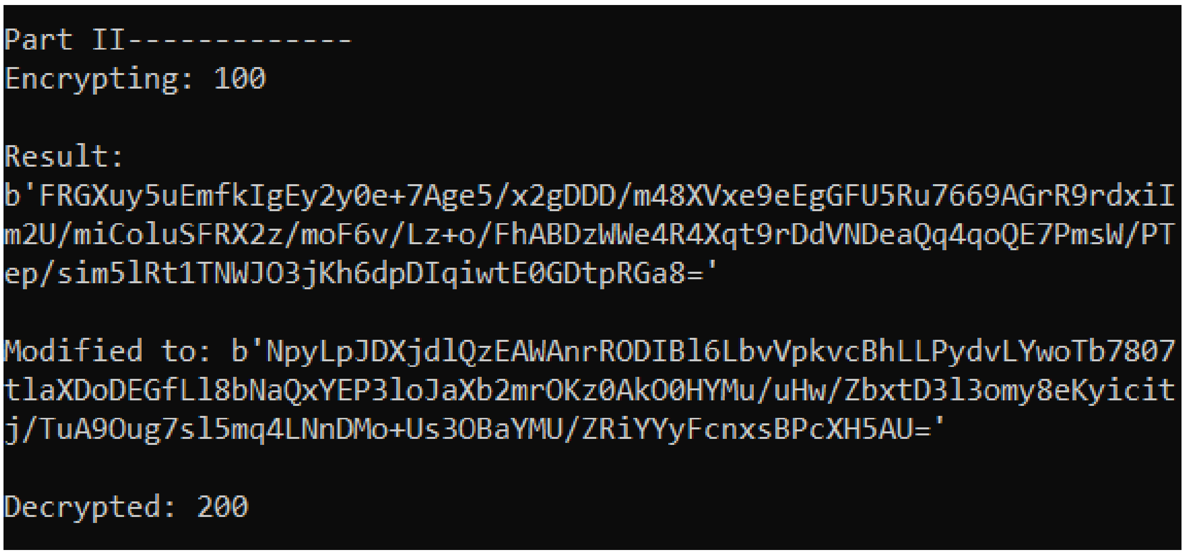
2.1 Results



```
(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>python pyrsa.py
Part I-------------
The message to encrypt is:
b'Hello World! This is Sherlock.\n'

The cipher text after encryption is:
b'NsV1XttACa3CN6Xm3DAM6eLTm2iKzFAtTm/tJ2a7qLmnKE/cAgatMf6p7I5zTNpfX54UqxoDIoRPLdeujj2gI+JRob6B/Bok/fcF/ozRlF8j9pNCXy0vKLELkvf3tGlIbuRyep9phwZgYQmVzRA+TikPD38S2XA/mKYHIKfdkFk='

The original message after decryption is:
bytearray(b'Hello World! This is Sherlock.\n')

The hash of the message is:
ed647cebd5d46b4911d546112af2d27c7b129d1c99db4a00c721a4c7be2042fc

The signed hash is:
b'5cjIKgThCTtB4XPW+EfUVhGMJsidzjioAwXw5aqUAEV9e3ElgOPRy8+sHVdWWER6LVe+t0EfCZM6cDtkd64hga773HcpgXBUBdln4hGCQzAadWABS1kSGTvb+xKHqcz+w01fLQRTrvzdLYbOsed+z0YiKgl+A8g2oEreiZcZFCM='

The original message after decrypting the hash:
 bytearray(b'ed647cebd5d46b4911d546112af2d27c7b129d1c99db4a00c721a4c7be2042fc')
```

3. Part 2

3.1 Results



```
Part II-------------
Encrypting: 100

Result:
b'FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiI
m2U/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq4qoQE7PmsW/PT
ep/sim5lRt1TNWJO3jKh6dpDIqiwtE0GDtpRGa8='

Modified to: b'NpyLpJDXjdlQzEAWAnrRODIBl6LbvVpkvcBhLLPydvLYwoTb7807
tlaXDoDEGfLl8bNaQxYEP3loJaXb2mrOKz0AkO0HYMu/uHw/ZbxtD3l3omy8eKyicit
j/TuA9Oug7sl5mq4LNnDMo+Us3OBaYMU/ZRiYYyFcnxsBPcXH5AU='

Decrypted: 200
```

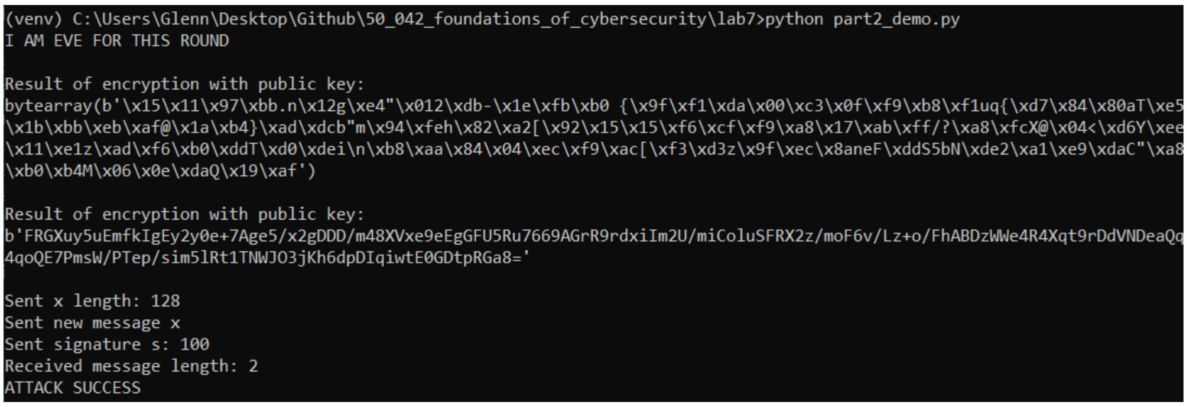
3.2 RSA Digital Signature Protocol Attack

For this lab I partnered Tan Yi Xuan (1002887)

For Alice's public key we used the provided key `mykey.pem.pub`. The code for this part is found in `part2_demo.py`

For the first round I played Eve's role

- Encrypt with the public key and print the `bytearray` version and the `b64encode`
- Send the `bytearray` version to my partner which happens to be 128 bytes long
- I also send the signature `s`
- My partner then did the same encryption with the same public key (Alice's) and checks that the digest is the same
- It was the same. He then sent me the length of his response (4 bytes) and the response to say that it was ok

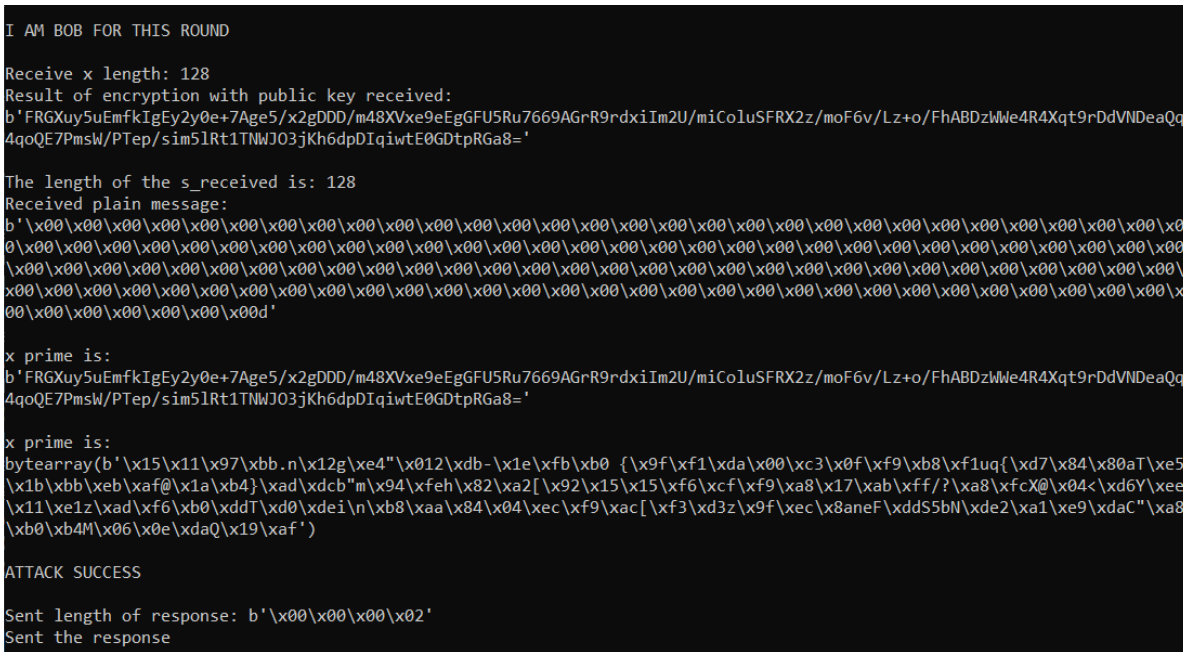


```
(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>python part2_demo.py
I AM EVE FOR THIS ROUND

Result of encryption with public key:
bytearray(b'\x15\x11\x97\xbb.n\x12g\xe4"\x012\xdb-\x1e\xfb\xb0 {\x9f\xf1\xda\x00\xc3\x0f\xf9\xb8\xf1uq{\xd7\x84\x80aT\xe5
\x1b\xbb\xeb\xaf@\x1a\xb4}\xad\xdcb"m\x94\xfeh\x82\xa2[\x92\x15\x15\xf6\xcf\xf9\xa8\x17\xab\xff/?\xa8\xfcX@\x04<\xd6Y\xee
\x11\xe1z\xad\xf6\xb0\xddT\xd0\xdei\n\xb8\xaa\x84\x04\xec\xf9\xac[\xf3\xd3z\x9f\xec\x8aneF\xddS5bN\xde2\xa1\xe9\xdaC"\xa8
\xb0\xb4M\x06\x0e\xdaQ\x19\xaf')

Result of encryption with public key:
b'FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiIm2U/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq
4qoQE7PmsW/PTep/sim5lRt1TNWJO3jKh6dpDIqiwtE0GDtpRGa8='

Sent x length: 128
Sent new message x
Sent signature s: 100
Received message length: 2
ATTACK SUCCESS
```

```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    '''
        For the first part I am Eve
        I will impersonate Alice and send x (message), s (signature)
        Bob already has Alice's public key, public key(e)
    '''
    print('I AM EVE FOR THIS ROUND\n')
    s = 100 #2019  # 111 11100011
    x = encrypt_RSA('mykey.pem.pub', pyrsa_sq_mul.pack_bigint(s))
    print('Result of encryption with public key:\n{}\n'.format(x))
    print('Result of encryption with public key:\n{}\n'.format(b64encode(x)))
    # Send x length
    sock.send(len(x).to_bytes(4, 'big'))
    print('Sent x length: {}'.format(len(x)))
    # Send x
    sock.sendall(x)
    print('Sent new message x')
    # Send s
    sock.send(len(pyrsa_sq_mul.pack_bigint(s)).to_bytes(4, 'big'))
    sock.sendall(pyrsa_sq_mul.pack_bigint(s))
    print('Sent signature s: {}'.format(s))
    # Receive acknowledgement from partner
    ## Receive the length of the message
    receive_length = int.from_bytes(sock.recv(4), 'big')
    print('Received message length: {}'.format(receive_length))
    ## Receive message
    receive_message = sock.recv(receive_length)
    if receive_message == b'OK':
        print('ATTACK SUCCESS\n')
    else:
        print('ATTACK FAILED\n')
```

In the second round I was Bob

- Partner encrypts with the public key
- I receive the message length (128 bytes)
- I print the `b64encode` version of the encrypted message
- I receive the length of the signature
- I receive the signature
- I then did the same encryption with the same public key (Alice's) and check that the digest is the same
- It was the same. I then sent the length of his response (4 bytes) and the response to say that it was ok



```
I AM BOB FOR THIS ROUND

Receive x length: 128
Result of encryption with public key received:
b'FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiIm2U/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq
4qoQE7PmsW/PTep/sim5lRt1TNWJO3jKh6dpDIqiwtE0GDtpRGa8='

The length of the s_received is: 128
Received plain message:
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00d'

x prime is:
b'FRGXuy5uEmfkIgEy2y0e+7Age5/x2gDDD/m48XVxe9eEgGFU5Ru7669AGrR9rdxiIm2U/miColuSFRX2z/moF6v/Lz+o/FhABDzWWe4R4Xqt9rDdVNDeaQq
4qoQE7PmsW/PTep/sim5lRt1TNWJO3jKh6dpDIqiwtE0GDtpRGa8='

x prime is:
bytearray(b'\x15\x11\x97\xbb.n\x12g\xe4"\x012\xdb-\x1e\xfb\xb0 {\x9f\xf1\xda\x00\xc3\x0f\xf9\xb8\xf1uq{\xd7\x84\x80aT\xe5
\x1b\xbb\xeb\xaf@\x1a\xb4}\xad\xdcb"m\x94\xfeh\x82\xa2[\x92\x15\x15\xf6\xcf\xf9\xa8\x17\xab\xff/?\xa8\xfcX@\x04<\xd6Y\xee
\x11\xe1z\xad\xf6\xb0\xddT\xd0\xdei\n\xb8\xaa\x84\x04\xec\xf9\xac[\xf3\xd3z\x9f\xec\x8aneF\xddS5bN\xde2\xa1\xe9\xdaC"\xa8
\xb0\xb4M\x06\x0e\xdaQ\x19\xaf')

ATTACK SUCCESS

Sent length of response: b'\x00\x00\x00\x02'
Sent the response
```

```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((HOST, PORT))
        '''
        For the second part I am Bob
        I will receive x (message), s (signature)
        I already has Alice's public key, public key(e)
        '''
        print('I AM BOB FOR THIS ROUND\n')
        #x_receive_length = sock.recv(4)
        x_receive_length = int.from_bytes(sock.recv(4), 'big')
        print('Receive x length: {}'.format(x_receive_length))
        x_receive = sock.recv(x_receive_length)
        print('Result of encryption with public key
received:\n{}\n'.format(b64encode(x_receive)))
        s_receive_length = int.from_bytes(sock.recv(4), 'big')
        print('The length of the s_received is:
{}'.format(s_receive_length))
        s_receive = sock.recv(s_receive_length)
        # s_receive_raw = int.from_bytes(s_receive, 'big')
        print('Received plain message:\n{}\n'.format(s_receive))
        s_receive = int.from_bytes(s_receive, 'big')
        s_receive = pyrsa_sq_mul.pack_bigint(s_receive)
        x_receive_prime = encrypt_RSA('mykey.pem.pub', s_receive)
        print('x prime is:\n{}\n'.format(b64encode(x_receive_prime)))
        print('x prime is:\n{}\n'.format(x_receive_prime))
        if x_receive == x_receive_prime:
            response_bob = 'OK'
            print('ATTACK SUCCESS\n')
        else:
            response_bob = 'ERR'
            print('ATTACK FAILED\n')
        sock.send(len(response_bob).to_bytes(4, 'big'))
        print('Sent length of response: {}'.format(len(response_bob).to_bytes(4, 'big')))
        sock.send(bytearray(response_bob, encoding='utf8'))
        print('Sent the response')
```

Conclusion

- In both attempts the attack was successful which was unsurprising as we were encrypting using a standard square multiply function with the same public key

3.3 Limitation of protocol attack

The limitation of the protocol attack is that it is difficult for Eve to trick Alice and Bob successfully by sending a valid ciphertext that Bob can decrypt without suspicion. It is easy for Eve to change the cipher which changes the plain text but if the decrypted plain text does not make sense then Bob will know that he is under attack (If Alice is the one sending a message to Bob)

For example, in the above case, Alice sends the number 100 to Bob (encrypting it before sending). If Eve knows the message, she can encrypt another number 2 and do a multiplication. When Bob decrypts it, he will see that the result is an integer which is the data type he is expecting and will not be suspicious that there has been an attack.

However, if Eve did not know that Alice has sent 100, Eve could encrypt maybe a phrase "hello Glenn" and do a multiplication. When Bob decrypts this, he will see a weird plain text and will suspect that there is an attacker.

4. Part 3

4.1 Generate RSA key

Create key

```python
from Crypto.PublicKey import RSA

key = RSA.generate(1024)
```

Save public and private keys

```python
f_public = open('part3_public.pem','wb')
f_private = open('part3_private.pem','wb')
publickey = key.publickey().exportKey('PEM')
privatekey = key.exportKey('PEM')
f_public.write(publickey)
f_private.write(privatekey)
f_public.close()
f_private.close()
```

4.2 Encrypt and Decrypt RSA - Crypto.Cipher.PKCS1_OAEP

The example provided was for encrypting and decrypting a session key

```python
# Encrypt the session key with the public RSA key
cipher_rsa = PKCS1_OAEP.new(recipient_key)
enc_session_key = cipher_rsa.encrypt(session_key)
# Decrypt the session key with the private RSA key
cipher_rsa = PKCS1_OAEP.new(private_key)
session_key = cipher_rsa.decrypt(enc_session_key)
```

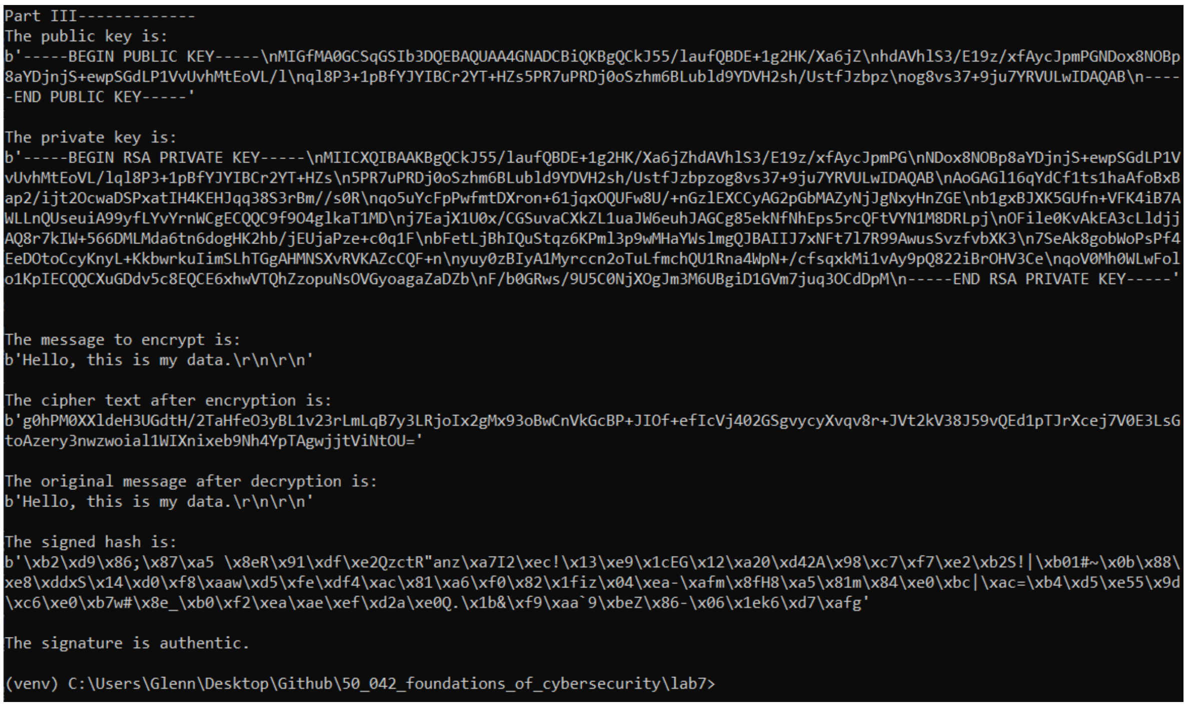
4.3 Sign and Verify signature - Crypto.Signature.PKCS1_PSS

Link: https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_pss.html

```python
from Crypto.Signature import PKCS1_PSS
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto import Random

# Signing
message = 'To be signed'
key = RSA.import_key(open('privkey.der').read())
h = SHA256.new(message)
signature = PKCS1_PSS.new(key).sign(h)

# Verification
```

```python
key = RSA.import_key(open('pubkey.der').read())
h = SHA256.new(message)
verifier = PKCS1_PSS.new(key)
try:
    verifier.verify(h, signature)
    print "The signature is authentic."
except (ValueError, TypeError):
    print "The signature is not authentic."
```

4.4 Results



```
Part III------------
The public key is:
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCkJ55/laufQBDE+1g2HK/Xa6jZ\nhdAVhlS3/E19z/xfAycJpmPGNDox8NOBp
8aYDjnjS+ewpSGdLP1VvUvhMtEoVL/l\nql8P3+1pBfYJYIBCr2YT+HZs5PR7uPRDj0oSzhm6BLubld9YDVH2sh/UstfJzbpz\nog8vs37+9ju7YRVULwIDAQAB\n----
-END PUBLIC KEY-----'

The private key is:
b'-----BEGIN RSA PRIVATE KEY-----\nMIICXQIBAAKBgQCkJ55/laufQBDE+1g2HK/Xa6jZhdAVhlS3/E19z/xfAycJpmPG\nNDox8NOBp8aYDjnjS+ewpSGdLP1V
vUvhMtEoVL/lql8P3+1pBfYJYIBCr2YT+HZs\n5PR7uPRDj0oSzhm6BLubld9YDVH2sh/UstfJzbpzog8vs37+9ju7YRVULwIDAQAB\nAoGAGl16qYdCf1ts1haAfoBxB
ap2/ijt2OcwaDSPxatIH4KEHJqq38S3rBm//s0R\nqo5uYcFpPwfmtDXron+61jqxOQUFw8U/+nGzlEXCCyAG2pGbMAZyNjJgNxyHnZGE\nb1gxBJXK5GUfn+VFK4iB7A
WLLnQUseuiA99yfLYvYrnWCgECQQC9f9O4glkaT1MD\nj7EajX1U0x/CGSuvaCXkZL1uaJW6euhJAGCg85ekNfNhEps5rcQFtVYN1M8DRLpj\nOFile0KvAkEA3cLldjj
AQ8r7kIW+566DMLMda6tn6dogHK2hb/jEUjaPze+c0q1F\nbFetLjBhIQuStqz6KPml3p9wMHaYWslmgQJBAIIJ7xNFt7l7R99AwusSvzfvbXK3\n7SeAk8gobWoPsPf4
EeDOtoCcyKnyL+KkbwrkuIimSLhTGgAHMNSXvRVKAZcCQF+n\nyuy0zBIyA1Myrccn2oTuLfmchQU1Rna4WpN+/cfsqxkMi1vAy9pQ822iBrOHV3Ce\nqoV0Mh0WLwFol
o1KpIECQQCXuGDdv5c8EQCE6xhwVTQhZzopuNsOVGyoagaZaDZb\nF/b0GRws/9U5C0NjXOgJm3M6UBgiD1GVm7juq3OCdDpM\n-----END RSA PRIVATE KEY-----'


The message to encrypt is:
b'Hello, this is my data.\r\n\r\n'

The cipher text after encryption is:
b'g0hPM0XXldeH3UGdtH/2TaHfeO3yBL1v23rLmLqB7y3LRjoIx2gMx93oBwCnVkGcBP+JIOf+efIcVj402GSgvycyXvqv8r+JVt2kV38J59vQEd1pTJrXcej7V0E3LsG
toAzery3nwzwoial1WIXnixeb9Nh4YpTAgwjjtViNtOU='

The original message after decryption is:
b'Hello, this is my data.\r\n\r\n'

The signed hash is:
b'\xb2\xd9\x86;\x87\xa5 \x8eR\x91\xdf\xe2QzctR"anz\xa7I2\xec!\x13\xe9\x1cEG\x12\xa20\xd42A\x98\xc7\xf7\xe2\xb2S!|\xb01#~\x0b\x88\
xe8\xddxS\x14\xd0\xf8\xaaw\xd5\xfe\xdf4\xac\x81\xa6\xf0\x82\x1fiz\x04\xea-\xafm\x8fH8\xa5\x81m\x84\xe0\xbc|\xac=\xb4\xd5\xe55\x9d
\xc6\xe0\xb7w#\x8e_\xb0\xf2\xea\xae\xef\xd2a\xe0Q.\x1b&\xf9\xaa`9\xbeZ\x86-\x06\x1ek6\xd7\xafg'

The signature is authentic.

(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>
```

4.5 Public - private key encryption - decryption

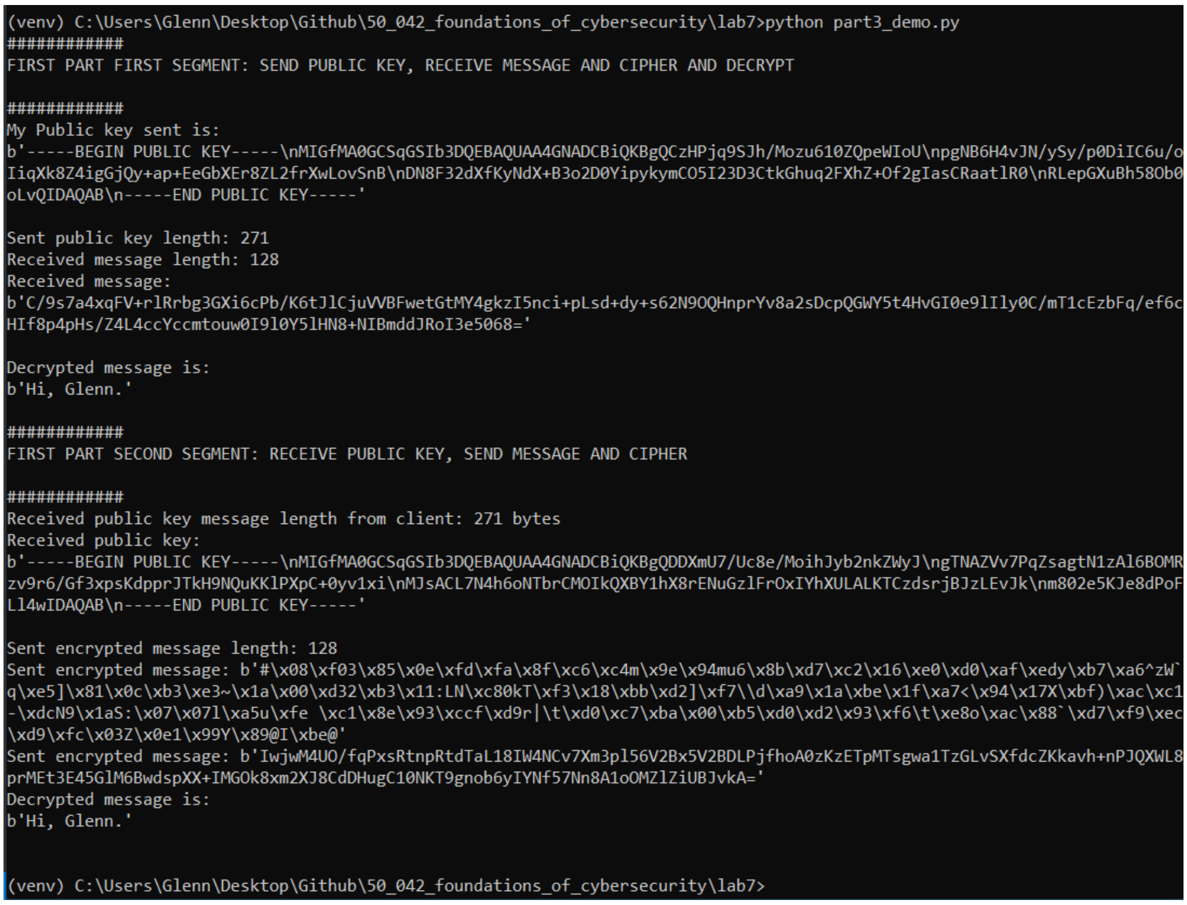
For the first round I sent my public key. I then received an encrypted message which I decrypted

For the second round I received my partner's public key. I encrypted a message and sent it to him. He verified it.

The code is as shown below which has the logic printed out as well. The code can be found in `part3_demo.py`

```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    print('############')
    print('FIRST PART FIRST SEGMENT: SEND PUBLIC KEY, RECEIVE MESSAGE AND 
CIPHER AND DECRYPT\n')
    print('############')
    # SEND THE PUBLIC KEY
    f_public = open('part3.pem.pub','rb')
    my_public_key = f_public.read()
    f_public.close()
    print('My Public key sent is:\n{}\n'.format(my_public_key))
    sock.send(len(my_public_key).to_bytes(4, 'big'))
    print('Sent public key length: {}'.format(len(my_public_key)))
    sock.sendall(my_public_key)
    # RECEIVE THE MESSAGE AND DECRYPT
    receive_length_segment1 = int.from_bytes(sock.recv(4), 'big')
    print('Received message length: {}'.format(receive_length_segment1))
    receive_message_segment1 = sock.recv(receive_length_segment1)
    print('Received message:\n{}\n'.format(b64encode(receive_message_segment1)))
    decrypted_message_segment1 = decrypt_RSA('part3.pem.priv', receive_message_segment1, 'part3')
    print('Decrypted message is:\n{}\n'.format(decrypted_message_segment1))
    time.sleep(1)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    print('############')
    print('FIRST PART SECOND SEGMENT: RECEIVE PUBLIC KEY, SEND MESSAGE AND CIPHER\n')
    print('############')
    # get public key message length from client
    pubkey_length = sock.recv(4)
    pubkey_length = int.from_bytes(pubkey_length, 'big')
    print("Received public key message length from client:", pubkey_length, 'bytes')
    # get public key from client
    pubkey = sock.recv(pubkey_length)
    with open('lab7_demo.pem.pub', 'wb') as pubkeyfile:
        pubkeyfile.write(pubkey)
    print("Received public key:\n{}\n".format(pubkey))
    # prepare message
    sent_mesage_segment1 = b'Hi, Glenn.'
    encrypted_message_segment1 = encrypt_RSA('lab7_demo.pem.pub', sent_mesage_segment1, 'part3')
    encrypted_message_segment1_length = len(encrypted_message_segment1).to_bytes(4, 'big')
    # send message length
    sock.send(encrypted_message_segment1_length)
    print('Sent encrypted message length: {}'.format(len(encrypted_message_segment1)))
    # send message
    sock.send(encrypted_message_segment1)
    print("Sent encrypted message: {}".format((encrypted_message_segment1)))
    print("Sent encrypted message: {}".format(b64encode(encrypted_message_segment1)))
    decrypted_message_segment2 = decrypt_RSA('mykey.pem.priv', encrypted_message_segment1, 'part3')
    print('Decrypted message is:\n{}\n'.format(decrypted_message_segment2))
    time.sleep(1)
```

Results



```
(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>python part3_demo.py
############
FIRST PART FIRST SEGMENT: SEND PUBLIC KEY, RECEIVE MESSAGE AND CIPHER AND DECRYPT

############
My Public key sent is:
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCzHPjq9SJh/Mozu610ZQpeWIoU\npgNB6H4vJN/ySy/p0DiIC6u/o
IiqXk8Z4igGjQy+ap+EeGbXEr8ZL2frXwLovSnB\nDN8F32dXfKyNdX+B3o2D0YipykymCO5I23D3CtkGhuq2FXhZ+Of2gIasCRaatlR0\nRLepGXuBh58Ob0
oLvQIDAQAB\n-----END PUBLIC KEY-----'

Sent public key length: 271
Received message length: 128
Received message:
b'C/9s7a4xqFV+rlRrbg3GXi6cPb/K6tJlCjuVVBFwetGtMY4gkzI5nci+pLsd+dy+s62N9OQHnprYv8a2sDcpQGWY5t4HvGI0e9lIly0C/mT1cEzbFq/ef6c
HIf8p4pHs/Z4L4ccYccmtouw0I9l0Y5lHN8+NIBmddJRoI3e5068='

Decrypted message is:
b'Hi, Glenn.'

############
FIRST PART SECOND SEGMENT: RECEIVE PUBLIC KEY, SEND MESSAGE AND CIPHER

############
Received public key message length from client: 271 bytes
Received public key:
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDDXmU7/Uc8e/MoihJyb2nkZWyJ\ngTNAZVv7PqZsagtN1zAl6BOMR
zv9r6/Gf3xpsKdpprJTkH9NQuKKlPXpC+0yv1xi\nMJsACL7N4h6oNTbrCMOIkQXBY1hX8rENuGzlFrOxIYhXULALKTCzdsrjBJzLEvJk\nm802e5KJe8dPoF
Ll4wIDAQAB\n-----END PUBLIC KEY-----'

Sent encrypted message length: 128
Sent encrypted message: b'#\x08\xf03\x85\x0e\xfd\xfa\x8f\xc6\xc4m\x9e\x94mu6\x8b\xd7\xc2\x16\xe0\xd0\xaf\xedy\xb7\xa6^zW`
q\xe5]\x81\x0c\xb3\xe3~\x1a\x00\xd32\xb3\x11:LN\xc80kT\xf3\x18\xbb\xd2]\xf7\\d\xa9\x1a\xbe\x1f\xa7<\x94\x17X\xbf)\xac\xc1
-\xdcN9\x1aS:\x07\x07l\xa5u\xfe \xc1\x8e\x93\xccf\xd9r|\t\xd0\xc7\xba\x00\xb5\xd0\xd2\x93\xf6\t\xe8o\xac\x88`\xd7\xf9\xec
\xd9\xfc\x03Z\x0e1\x99Y\x89@I\xbe@'
Sent encrypted message: b'IwjwM4UO/fqPxsRtnpRtdTaL18IW4NCv7Xm3pl56V2Bx5V2BDLPjfhoA0zKzETpMTsgwa1TzGLvSXfdcZKkavh+nPJQXWL8
prMEt3E45GlM6BwdspXX+IMGOk8xm2XJ8CdDHugC10NKT9gnob6yIYNf57Nn8A1oOMZlZiUBJvkA='
Decrypted message is:
b'Hi, Glenn.'


(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>
```

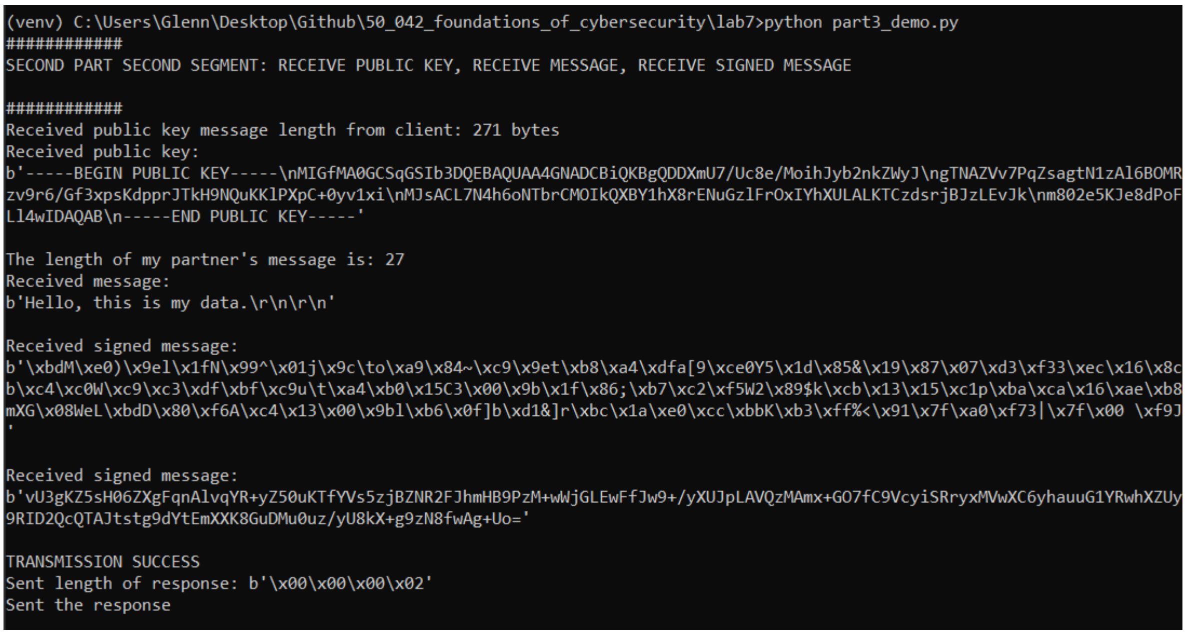
4.6 Sign and verify

For the first round I sent my public key, the plain message (read from the file `mydata.txt`) ant the signed hash (signed with my private key). My partner verified the signature and set a reply

For the second round I received my partner's public key, the plain message and the signed hash (signed with my partner's private key). I used the public key and the plain message to verify the signed hash.

The code is as shown below which has the logic printed out as well. The code can be found in `part3_demo.py`

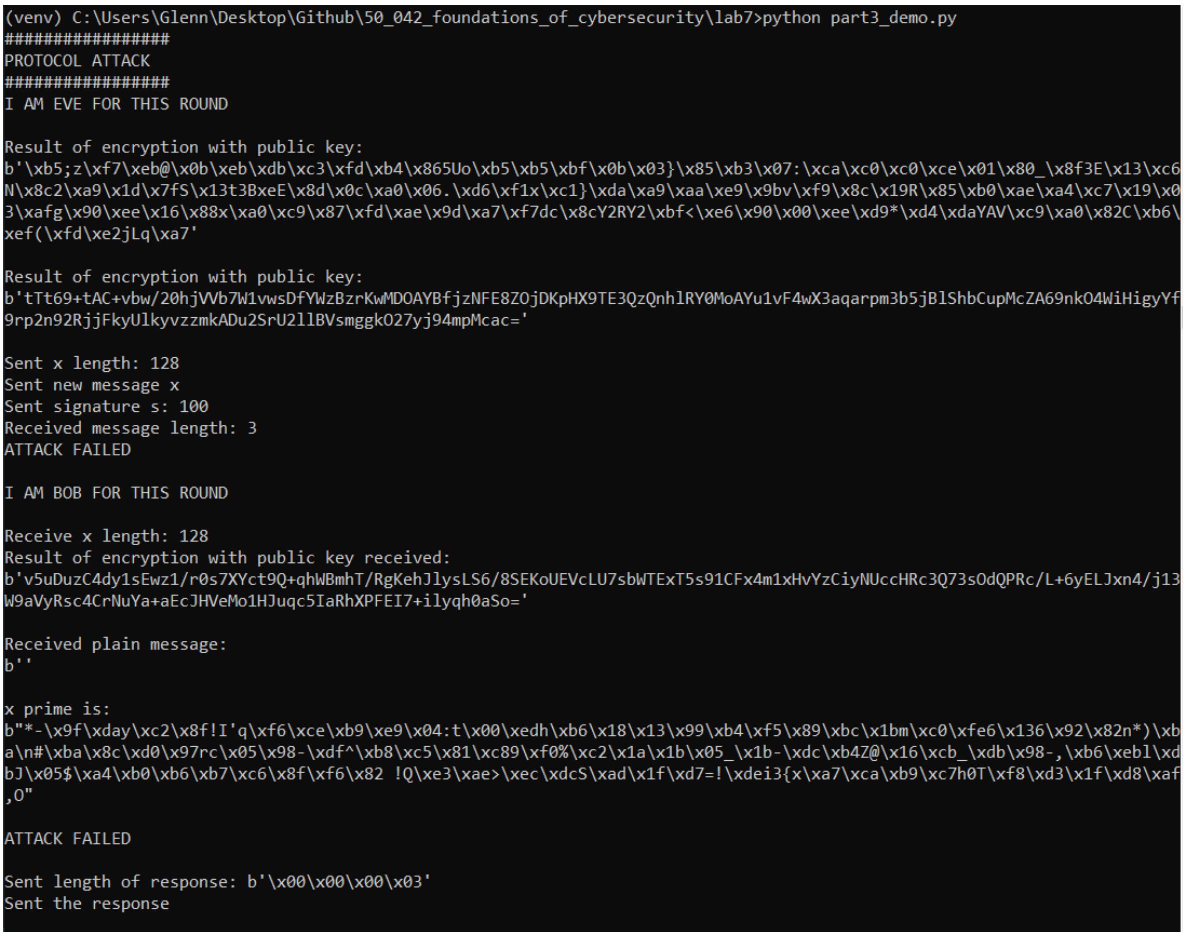
```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    '''
        For the second part I am sending my public key and message that is signed with my private key
        '''
    print('############')
    print('SECOND PART FIRST SEGMENT: SEND PUBLIC KEY, SEND MESSAGE, SEND SIGNED MESSAGE\n')
    print('############')
    # SEND THE PUBLIC KEY
    f_public = open('part3.pem.pub','rb')
    my_public_key = f_public.read()
    f_public.close()
    print('My Public key sent is:\n{}\n'.format(my_public_key))
    sock.send(len(my_public_key).to_bytes(4, 'big'))
    print('Sent public key length: {}'.format(len(my_public_key)))
    sock.sendall(my_public_key)
    # SIGN THE MESSAGE
    with open("mydata.txt", 'rb') as f_message_data:
        message_data = f_message_data.read()
    signed_hash, hashed_message = sign_RSA('part3.pem.priv', message_data)
    print('The signed hash is:\n{}\n'.format(signed_hash))
    # SEND THE MESSAGE LENGTH and MESSAGE
    sock.send(len(message_data).to_bytes(4, 'big'))
    print('Message length_sent is: {}'.format(len(message_data)))
    sock.send(message_data)
    print('Message sent is: {}'.format(message_data))
    # SEND THE SIGNED MESSAGE
    sock.send(signed_hash)
    print('Raw Signed Message sent is:\n{}\n'.format(signed_hash))
    print('Signed Message sent is:\n{}\n'.format(b64encode(signed_hash)))
    print('JUST TO CHECK {}'.format(verify_sign('part3.pem.pub', signed_hash, message_data)))
    # Receive acknowledgement from partner
    ## Receive the length of the message
    receive_length = int.from_bytes(sock.recv(4), 'big')
    print('Received message length: {}'.format(receive_length))
    ## Receive message
    receive_message = sock.recv(receive_length)
    if receive_message == b'OK':
        print('TRANSMISSION SUCCESS\n')
    else:
        print('TRANSMISSION FAILED\n')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    '''
        For the second part I am receiving my partner's public key and message that is signed with his private key
        '''
    print('############')
    print('SECOND PART SECOND SEGMENT: RECEIVE PUBLIC KEY, RECEIVE MESSAGE, RECEIVE SIGNED MESSAGE\n')
    print('############')
    # get public key message length from client
    pubkey_length = sock.recv(4)
    pubkey_length = int.from_bytes(pubkey_length, 'big')
    print("Received public key message length from client:", pubkey_length, 'bytes')
    # get public key from client
    pubkey = sock.recv(pubkey_length)
    with open('lab7_demo2.pem.pub', 'wb') as pubkeyfile:
        pubkeyfile.write(pubkey)
    print("Received public key:\n{}\n".format(pubkey))
    # Receive message length and message
    message_partner_length = int.from_bytes(sock.recv(4), 'big')
    print("The length of my partner's message is: {}".format(message_partner_length))
    message_partner = sock.recv(message_partner_length)
    print('Received message:\n{}\n'.format(message_partner))
    # Receive signed message
    signed_message_partner = sock.recv(128)
    print('Received signed message:\n{}\n'.format(signed_message_partner))
    print('Received signed message:\n{}\n'.format(b64encode(signed_message_partner)))
    if verify_sign('lab7_demo2.pem.pub', signed_message_partner, message_partner) == "The signature is authentic.":
        response = 'OK'
        print('TRANSMISSION SUCCESS')
    else:
        response = 'ERR'
        print('TRANSMISSION FAILED')
    sock.send(len(response).to_bytes(4, 'big'))
    print('Sent length of response: {}'.format(len(response).to_bytes(4, 'big')))
    sock.send(bytearray(response, encoding='utf8'))
    print('Sent the response')
```



```
(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>python part3_demo.py
############
SECOND PART SECOND SEGMENT: RECEIVE PUBLIC KEY, RECEIVE MESSAGE, RECEIVE SIGNED MESSAGE

############
Received public key message length from client: 271 bytes
Received public key:
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDDXmU7/Uc8e/MoihJyb2nkZWyJ\ngTNAZVv7PqZsagtN1zAl6BOMR
zv9r6/Gf3xpsKdpprJTkH9NQuKKlPXpC+0yv1xi\nMJsACL7N4h6oNTbrCMOIkQXBY1hX8rENuGzlFrOxIYhXULALKTCzdsrjBJzLEvJk\nm802e5KJe8dPoF
Ll4wIDAQAB\n-----END PUBLIC KEY-----'

The length of my partner's message is: 27
Received message:
b'Hello, this is my data.\r\n\r\n'

Received signed message:
b'\xbdM\xe0)\x9el\x1fN\x99^\x01j\x9c\to\xa9\x84~\xc9\x9et\xb8\xa4\xdfa[9\xce0Y5\x1d\x85&\x19\x87\x07\xd3\xf33\xec\x16\x8c
b\xc4\xc0W\xc9\xc3\xdf\xbf\xc9u\t\xa4\xb0\x15C3\x00\x9b\x1f\x86;\xb7\xc2\xf5W2\x89$k\xcb\x13\x15\xc1p\xba\xca\x16\xae\xb8
mXG\x08WeL\xbdD\x80\xf6A\xc4\x13\x00\x9bl\xb6\x0f]b\xd1&]r\xbc\x1a\xe0\xcc\xbbK\xb3\xff%<\x91\x7f\xa0\xf73|\x7f\x00 \xf9J
'

Received signed message:
b'vU3gKZ5sH06ZXgFqnAlvqYR+yZ50uKTfYVs5zjBZNR2FJhmHB9PzM+wWjGLEwFfJw9+/yXUJpLAVQzMAmx+GO7fC9VcyiSRryxMVwXC6yhauuG1YRwhXZUy
9RID2QcQTAJtstg9dYtEmXXK8GuDMu0uz/yU8kX+g9zN8fwAg+Uo='

TRANSMISSION SUCCESS
Sent length of response: b'\x00\x00\x00\x02'
Sent the response
```

4.7 Redo protocol attack with new RSA

The code is the same as the protocol attack from Part 2 except that we are using the new RSA encryption function which uses OAEP. This time, the attack fails because we are getting a different digest each time when we encrypt with the public key



```
(venv) C:\Users\Glenn\Desktop\Github\50_042_foundations_of_cybersecurity\lab7>python part3_demo.py
#################
PROTOCOL ATTACK
#################
I AM EVE FOR THIS ROUND

Result of encryption with public key:
b'\xb5;z\xf7\xeb@\x0b\xeb\xdb\xc3\xfd\xb4\x865Uo\xb5\xb5\xbf\x0b\x03}\x85\xb3\x07:\xca\xc0\xc0\xce\x01\x80_\x8f3E\x13\xc6
N\x8c2\xa9\x1d\x7fS\x13t3BxeE\x8d\x0c\xa0\x06.\xd6\xf1x\xc1}\xda\xa9\xaa\xe9\x9bv\xf9\x8c\x19R\x85\xb0\xae\xa4\xc7\x19\x0
3\xafg\x90\xee\x16\x88x\xa0\xc9\x87\xfd\xae\x9d\xa7\xf7dc\x8cY2RY2\xbf<\xe6\x90\x00\xee\xd9*\xd4\xdaYAV\xc9\xa0\x82C\xb6\
xef(\xfd\xe2jLq\xa7'

Result of encryption with public key:
b'tTt69+tAC+vbw/20hjVVb7W1vwsDfYWzBzrKwMDOAYBfjzNFE8ZOjDKpHX9TE3QzQnhlRY0MoAYu1vF4wX3aqarpm3b5jBlShbCupMcZA69nkO4WiHigyYf
9rp2n92RjjFkyUlkyvzzmkADu2SrU2llBVsmggkO27yj94mpMcac='

Sent x length: 128
Sent new message x
Sent signature s: 100
Received message length: 3
ATTACK FAILED

I AM BOB FOR THIS ROUND

Receive x length: 128
Result of encryption with public key received:
b'v5uDuzC4dy1sEwz1/r0s7XYct9Q+qhWBmhT/RgKehJlysLS6/8SEKoUEVcLU7sbWTExT5s91CFx4m1xHvYzCiyNUccHRc3Q73sOdQPRc/L+6yELJxn4/j13
W9aVyRsc4CrNuYa+aEcJHVeMo1HJuqc5IaRhXPFEI7+ilyqh0aSo='

Received plain message:
b''

x prime is:
b"*-\x9f\xday\xc2\x8f!I'q\xf6\xce\xb9\xe9\x04:t\x00\xedh\xb6\x18\x13\x99\xb4\xf5\x89\xbc\x1bm\xc0\xfe6\x136\x92\x82n*)\xb
a\n#\xba\x8c\xd0\x97rc\x05\x98-\xdf^\xb8\xc5\x81\xc89\xf0%\xc2\x1a\x1b\x05_\x1b-\xdc\xb4Z@\x16\xcb_\xdb\x98-,\xb6\xebl\xd
bJ\x05$\xa4\xb0\xb6\xb7\xc6\x8f\xf6\x82 !Q\xe3\xae>\xec\xdcS\xad\x1f\xd7=!\xdei3{x\xa7\xca\xb9\xc7h0T\xf8\xd3\x1f\xd8\xaf
,O"

ATTACK FAILED

Sent length of response: b'\x00\x00\x00\x03'
Sent the response
```

4.8 Explain the purpose of Optimal Asymmetric Encryption Padding (OAEP) to encrypt and decrypt using RSA. Explain how it works.

Useful references

- https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding
- https://medium.com/blue-space/improving-the-security-of-rsa-with-oaep-e854a5084918
- https://asecuritysite.com/encryption/rsa_oaep

Purpose of OAEP

OAEP serves to convert a deterministic encryption scheme which is the traditional RSA into a probabilistic scheme. It has proven to be secure against chosen plaintext attack.

High level description

OAEP is a form of feistel network which uses a pair of random oracles to process the plaintext prior to asymmetric encryption.

Essentially, it pads the message then encrypts it as $Enc = MessagePadded^e (mod\, n)$ and decrypt it as $MessagePadded = Enc^d (mod\, n)$

Low Level description

Some variable declarations

- n: RSA modulus
- k_0 and k_1 are chosen integers by OAEP
- m is plaintext which is $n - k_0 - k_1$ bits long
- G and H are random oracles
- $\oplus$

Encryption

Step 1

The message is padded with k_1 0s and will be $n - k_1$ bits long. Let this message be `m'`

Step 2

Generate a random string `r` which is k_0 bits long.

Step 3

Random oracle `G` expands the k_0 bits of `r` to $n - k_0$. Let this expanded string be `r'`

Step 4

Compute the XOR of `m'` and `r'`. $X = r' \oplus m'$

Step 5

Random Oracle `H` reduces the $n - k_0$ bits of `X` to k_0 bits. We call this $H(X)$

Step 6

Generate the right hand side output which is given by $Y = r \oplus H(X)$

Step 7

Take the X from Step 4 and the Y from Step 6 and concatenate them

Result = $X||Y$

Decryption

Step 1

Get the random string `r` given by $r = Y \oplus H(X)$.

Y is the right hand side output from Step 7 encryption. $H(X)$ is the result of the random oracle from Step 4 encryption

Step 2

Recover the message with the padded 0s

$m' = X \oplus r'$

Where X is the left hand side output from Step 7 encryption and r' is the result from Step 3 encryption.

Step 3

Remove the padding from `m'` to get back the original message m

4.9 Explain the purpose of Probabilistic Signature Scheme (PSS) to sign and verify using RSA. Explain how it works.

References

- https://en.wikipedia.org/wiki/Probabilistic_signature_scheme
- https://www.cryptosys.net/pki/manpki/pki_rsaschemes.html
- https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/whats_new/security_changes_80/security_whatsnew_sr1.html

Purpose

PSS was designed to allow modern methods of security analysis to prove that its security directly relates to that of the RSA program. PSS is more robust in theory than traditional RSA schemes.

PSS is a signature scheme with an appendix which requires the message itself to verify the signature though the message is not recoverable from the signature

PSS is randomized and will produce a different signature value each time

How it works

Encryption

PSS uses a hash function to be specified, a mask generation function

Step 1

Hash the message to be signed with a hash function (Most use SHA-1 as default)

Step 2

The hash is transformed into an encoded message. This transformation operation uses padding which is much more random. This is where the mask generation function is used.

Step 3

A signature primitive is applied to the encoded message by using the private key to produce the signature

Verification

Step 1

Hash the message to be signed with a hash function (Most use SHA-1 as default). Must be the same hash function as the encryption part

Step 2

A verification primitive is then applied to the signature by using the public key of the key pair to recover the message.

Step 3

Verify that the encoded message is a valid transformation of the hash value