Experiment 09

Write a Program to implement Digital signature scheme using RSA

Roll No.	19
Name	Manav Jawrani
Class	D15-A
Subject	Security Lab
LO Mapped	LO1: To apply the knowledge of symmetric cryptography to implement classical ciphers. LO2 : Demonstrate Key management, distribution and user authentication.

Aim: Write a Program to implement Digital signature scheme using RSA.

Introduction:

RSA:

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes, the Public Key is given to everyone and the Private key is kept private.

An example of asymmetric cryptography:
A client (for example browser) sends its public key to the server and requests for some data.
The server encrypts the data using the client's public key and sends the encrypted data. Client receives this data and decrypts it. Since this is asymmetric, nobody else except the browser can decrypt the data even if a third party has the public key of the browser.
The idea! The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is multiplication of two large prime numbers. And private keys are also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024 bit keys could be broken in the near future. But till now it seems to be an infeasible task.

Algorithm of RSA: -
Generating Public Key:
Select two prime no's. Suppose P = 53 and Q = 59.
Now first part of the Public key: n = P*Q = 3127.
We also need a small exponent say e :
But e Must be an integer.

Not be a factor of n.
$1 < e < \Phi(n)$
Let us now consider it to be equal to 3.
Our Public Key is made of n and e

Generating Private Key:
We need to calculate Φ(n):
Such that Φ(n) = (P-1)(Q-1)
so, Φ(n) = 3016

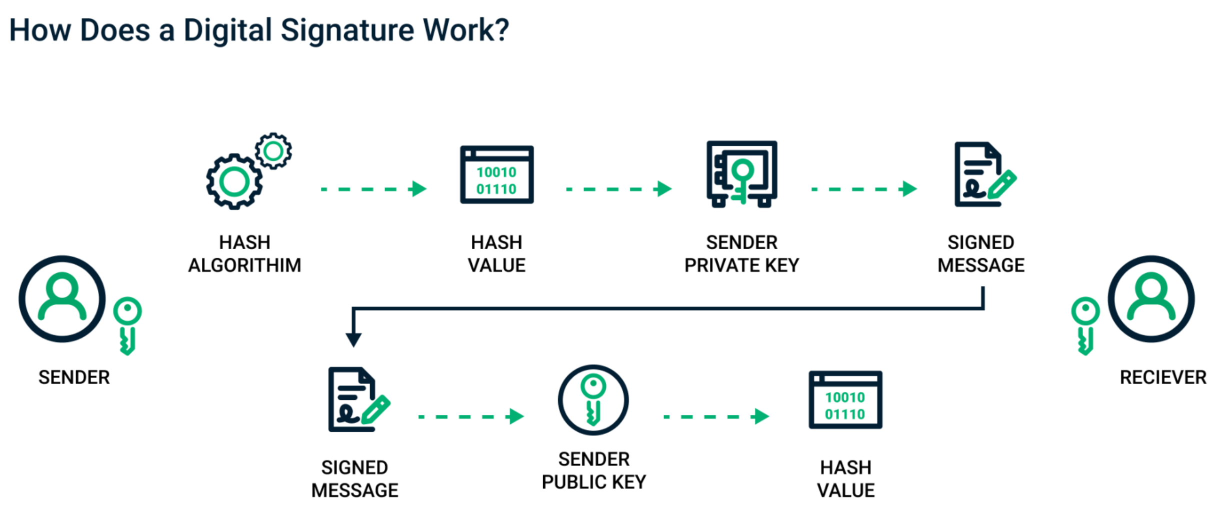
Now calculate Private Key, d :
d = (k*Φ(n) + 1) / e for some integer k
For k = 2, the value of d is 2011.
Now we are ready with our – Public Key (n = 3127 and e = 3) and Private Key(d = 2011)

Now we will encrypt “HI” :
Convert letters to numbers : H = 8 and I = 9

Thus Encrypted Data c = 89e mod n.
Thus our Encrypted Data comes out to be 1394

Now we will decrypt 1394 :
Decrypted Data = cd mod n.
Thus our Encrypted Data comes out to be 89
8 = H and I = 9 i.e. "HI".

What is Digital Signature?
A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document. It's the digital equivalent of a handwritten signature or stamped seal, but it offers far more inherent security. A digital signature is intended to solve the problem of tampering and impersonation in digital communications.
Digital signatures can provide evidence of origin, identity and status of electronic documents, transactions or digital messages. Signers can also use them to acknowledge informed consent.



Algorithm:

Keeping the image above in mind, go ahead and see how the entire process works, starting from creating the key pair to verifying the signature at the end.

1. Key Generation

There are two steps in the key generation process: parameter generation and per-user keys.

Parameter Generation

- Initially a user needs to choose a cryptographic hash function (H) along with output length in bits |H|. Modulus length N is used in when output length |H| is greater.
- Then choose a key length L where it should be multiple of 64 and lie in between 512 and 1024 as per Original DSS length. However, lengths 2048 or 3072 are recommended by NIST for lifetime key security.
- The values of L and N need to be chosen in between (1024, 60), (2048, 224), (2048, 256), or (3072, 256) according to FIPS 186-4. Also, a user should chose modulus length N in such a way that modulus length N should be less than key length (N<L) and less than and equal to output length (N<=|H|).
- Later a user can choose a prime number q of N bit and another prime number as p of L bit in such a way that p-1 is multiple of q. And then choose h as an integer from the list (2……..p-2).
- Once you get p and q values, find out

g = h^(p-1)/q*mod(p). If you get g = 1, please try another value for h and compute again for g except 1.
p, q and g are the algorithm parameters that are shared amongst different users of the systems.
Per-user Keys
To compute the key parameters for a single user, first choose an integer x (private key) from the list (1…….q-1), then compute the public key, y=g^(x)*mod(p).
2. Signature Generation

- It passes the original message (M) through the hash function (H#) to get our hash digest(h).
- It passes the digest as input to a signing function, whose purpose is to give two variables as output, s, and r.
- Apart from the digest, you also use a random integer k such that 0 < k < q.
- To calculate the value of r, you use the formula r = (gk mod p) mod q.
- To calculate the value of s, you use the formula s = [K-1(h+x . R)mod q].
- It then packages the signature as {r,s}.
- The entire bundle of the message and signature {M,r,s} are sent to the receiver.

3. Key Distribution
While distributing keys, a signer should keep the private key (x) secret and publish the public key (y) and send the public key (y) to the receiver without any secret mechanism.
Signing
Signing of message m should be done as follows:

- first choose an integer k from (1……q-1)
- compute

r = g^(k)*mod(p)*mod(q). If you get r = 0, please try another random value of k and compute again for r except 0.

- Calculate

s=(k^(-1)*(H(m)+xr))*mod(q). If you get s = 0, please try another random value of k and compute again for s except 0.

- The signature is defined by two key elements (r,s). Also, key elements k and r are used to create a new message. Nevertheless, computing r with modular exponential process is a very expensive process and computed before the message is known. Computation is done with the help of the Euclidean algorithm and Fermat's little theorem.

4. Signature Verification

- You use the same hash function (H#) to generate the digest h.
- You then pass this digest off to the verification function, which needs other variables as parameters too.
- Compute the value of w such that: s*w mod q = 1
- Calculate the value of u1 from the formula, u1 = h*w mod q
- Calculate the value of u2 from the formula, u2 = r*w mod q
- The final verification component v is calculated as v = [((gu1 . yu2) mod p) mod q].
- It compares the value of v to the value of r received in the bundle.
- If it matches, the signature verification is complete.

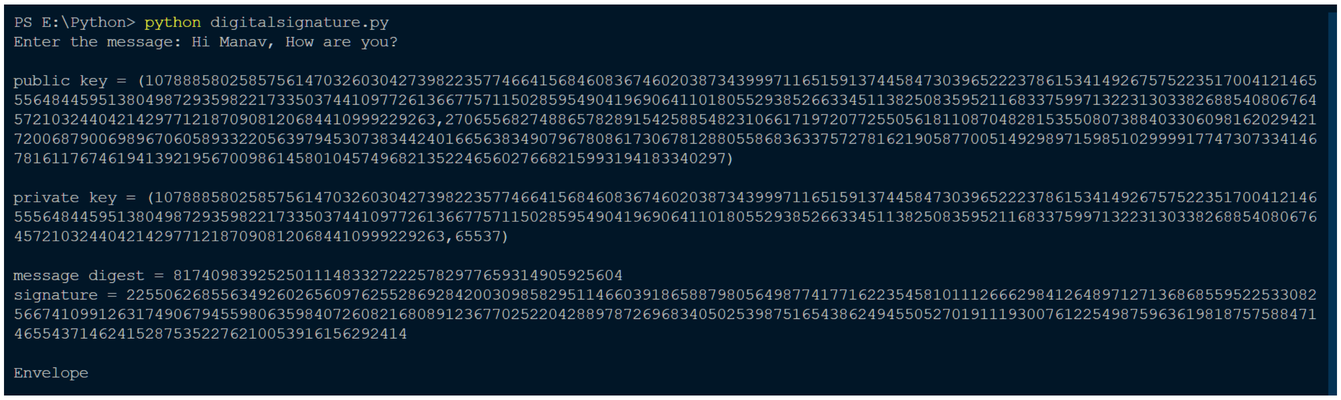
Implementation:

Code:

```
from hashlib import sha1
from Crypto.PublicKey import RSA
keypair = RSA.generate(bits=1024)
message = input("Enter the message: ")
algorithm = sha1()
algorithm.update(bytes(message,'utf-8'))
digest = int.from_bytes(algorithm.digest(),byteorder='little')
print(f'\npublic key = ({keypair.n},{keypair.d})')
print(f'\nprivate key = ({keypair.n},{keypair.e})')
print(f'\nmessage digest = {digest}')
signature = pow(digest,keypair.e,keypair.n)
print(f'signature = {signature}')
print('\nEnvelope')
print(f'\nmessage: {message}')
print(f'\nsignature = {signature}')
print('\nMESSAGE SENT')
rec_message = message
rec_signature = signature
algorithm = sha1()
algorithm.update(bytes(rec_message,'utf-8'))
```

```
gen_digest = int.from_bytes(algorithm.digest(),byteorder='little')
digest_decrypt = pow(rec_signature,keypair.d,keypair.n)
print(f'\nReceived Message: {rec_message}')
print(f'\Received Signature: {rec_signature}')
print(f'\nDigest generated from message: {gen_digest}')
print(f'\nDigest generated from signature: {digest_decrypt}')
if gen_digest == digest_decrypt:
  {
     print('\nMessage is not tampered')
  }
else:
  {
   print('\nMessage is tampered')
  }
```

Output:



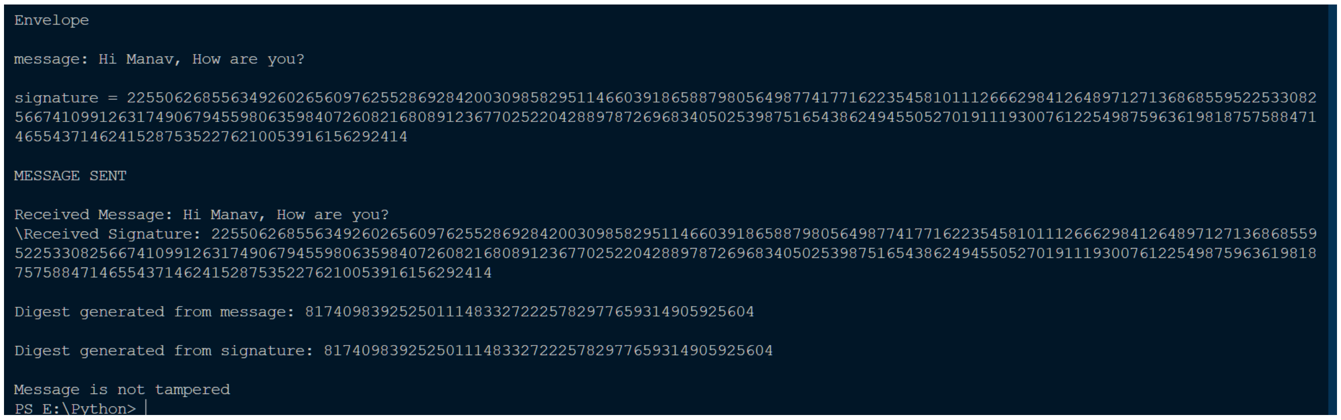
```
PS E:\Python> python digitalsignature.py
Enter the message: Hi Manav, How are you?

public key = (10788858025857561470326030427398223577466415684608367460203873439997116515913744584730396522237861534149267575223517004121465
5564844595138049872935982217335037441097726136677571150285954904196906411018055293852663345113825083595211683375997132231303382688540806764
572103244042142977121870908120684410999229263,270655682748865782891542588548231066171972077255056181108704828153550807388403306098162029421
7200687900698967060589332205639794530738344240166563834907967808617306781288055868363375727816219058770051492989715985102999917747307334146
7816117674619413921956700986145801045749682135224656027668215993194183340297)

private key = (1078885802585756147032603042739822357746641568460836746020387343999711651591374458473039652223786153414926757522351700412146
5556484459513804987293598221733503744109772613667757115028595490419690641101805529385266334511382508359521168337599713223130338268854080676
4572103244042142977121870908120684410999229263,65537)

message digest = 817409839252501114833272225782977659314905925604
signature = 2255062685563492602656097625528692842003098582951146603918658879805649877417716223545810111266629841264897127136868559522533082
56674109912631749067945598063598407260821680891236770252204288978726968340502539875165438624945505270191111930076122549875963619818757588471
465543714624152875352276210053916156292414

Envelope
```



```
Envelope

message: Hi Manav, How are you?

signature = 2255062685563492602656097625528692842003098582951146603918658879805649877417716223545810111266629841264897127136868559522533082
56674109912631749067945598063598407260821680891236770252204288978726968340502539875165438624945505270191111930076122549875963619818757588471
465543714624152875352276210053916156292414

MESSAGE SENT

Received Message: Hi Manav, How are you?
\Received Signature: 2255062685563492602656097625528692842003098582951146603918658879805649877417716223545810111266629841264897127136868559
52253308256674109912631749067945598063598407260821680891236770252204288978726968340502539875165438624945505270191111930076122549875963619818
757588471465543714624152875352276210053916156292414

Digest generated from message: 817409839252501114833272225782977659314905925604

Digest generated from signature: 817409839252501114833272225782977659314905925604

Message is not tampered
PS E:\Python> |
```

<u>**Conclusion**</u>:

Thus, we have implemented the Digital Signature concept using code.