

Computer Networks Laboratory – Assignment 11

Subhash S – 111801042

Computer Science and Engineering, IIT Palakkad

Generating public and private keys for users A and B

File: q1.py and utils.py

Aim

To generate public, private key pairs for users A and B.

Functions

1. *is_prime*: Given a number n it checks if the number is prime.
2. *generate_prime*: Give the number of bits n , it generates a random n bit prime number.
3. *read_public_key* and *read_private_key*: Used to read public and private keys of a user respectively.

Program Flow

1. I generate two large prime numbers p and q of size 512 bits each, this ensures that the modulus $n = p \times q$ comes to 1024 bits.
2. Now I store the modulus $p \times q$ in the local variable n .
3. I compute the totient of n i.e., $\phi(n) = (p - 1) \times (q - 1)$
4. Now I find the public exponent ($e: 5 < e < \phi(n)$) iteratively by starting from 5 and incrementing by 1 until it is relatively prime to the totient computed in 3.
5. I find the modular inverse of $e \equiv \phi(n)$
6. Now the public key pair is (e, n) and the private key pair is (d, n) .

Figure 1: The contents of all the four files after q1.py was run

```

$ /mnt/i/Networks-Lab/lab11 > python3 q1.py
Private and public keys for A successfully generated.
Private and public keys for B successfully generated.
$ /mnt/i/Networks-Lab/lab11 > echo | cat A.pub -
396271245687989827762042486591473348143968655741373404628560382603281551304455093546782951729061401631601432025638143017664563217764960649983423013421373618
76269989853639757872055725493437261813148246057871604634486805308762614569976089271290258573838903483200262822928131296179838491333075885508655441802161 660
452376146648379603404144319122246906614426235622341047680637672135918840758489244638252881769002719335720042730238362774272029608267749972371689035622861072
9778394058392148592997689086536483547042907865550303871435270482962421514853448222796765822098081311573706987731135312580619803541046265021572606973
$ /mnt/i/Networks-Lab/lab11 > echo | cat A.pri -
5 6604523761466483796034041443191222469066144262356223410476806376721359188407584892446382528817690027193357200427302383627742720296082677499723716890356228
610729778394050392148592997689086536483547042907865550303871435270482962421514853448222796765822098081311573706987731135312580619803541046265021572606973
$ /mnt/i/Networks-Lab/lab11 > echo | cat B.pub -
363972533237134705723493256478945218126192068219333587697385254385865043212139971466832726520476053758167920627985793489529915269301800651652626994907662305
46161003562316793490645353141601975753266907256781139427340398616709203443297501240174905683536024895173109400911895101326626885940988590751106169657843 849
269244219980980021484265117538842294448159178445037960565593380351767494993266755943028547777458769058481465300184808903135628370868187189462988117878902201
67998783441795634837102236105263515878685492850722605652183898715425597632501809128480198000458490746928373808349339318673708593180714222825924940437
$ /mnt/i/Networks-Lab/lab11 > echo | cat B.pri -
7 8492692442199809800214842651175388422944481591784450379605655933803517674949932667559430285477774587690584814653001848089031356283708681871894629881178789
0220167998783441795634837102236105263515878685492850722605652183898715425597632501809128480198000458490746928373808349339318673708593180714222825924940437
$ /mnt/i/Networks-Lab/lab11 >
```

Sign and encrypt the message using the keys generated in q1

File: q2.py and utils.py

Aim

To sign and encrypt the message using the keys generated in the previous questions so that it can be sent to be in a secure manner.

Functions

1. *msg2blocks*: Splits any message of arbitrary length into blocks of size $n/16$ bytes each.
2. *str2int*: Given a string *s*, it converts the ASCII string into an integer by converting each character from ASCII to its integer value.
3. *encrypt*: This function is used to encrypt the message *m* by performing the operation $m^e \equiv n$ where *m* is the message, *e* is the public exponent and *n* is the modulus.
4. *sign*: This function is used to sign the message by performing the operation $s^d \equiv n$, where *s* is the message digest and *d*, is the secret exponent.

Program

1. I read the private key of A and the public key of B since they are used in encrypting the message and signing the message respectively.
2. I read the message from the file *message.txt*.
3. Now I append the public key tuple of B to the message above, delimited by a colon, this way of constructing the message allows B to decrypt and verify the signature.
4. Since the message can be of any length, I divide it into blocks of appropriate size using the *msg2blocks* function mentioned above.
5. For each of the blocks,
 - a. I sign the block using the *sign* function mentioned above. This uses the private key of A.
 - b. I encrypt the previously signed block using the *encrypt* function mentioned above. This uses the public key of B.
 - c. I append this encrypted message to the variable *encrypted_msg* which accumulates the final encrypted message from each of the blocks.
6. Now the encryption and signing are completed.
7. I write the signed and encrypted message to the file *secret.txt*.

Figure 2: Shows the content of secret.txt after running q2.py

```
/mnt/i/Networks-Lab/lab11 > python3 q2.py
Signed and encrypted message written to secret.txt
/mnt/i/Networks-Lab/lab11 > echo | cat secret.txt -
155492081328423759505342230734020969470427071800709244071410452442841981869158127914842540623239128627082114068630183983993577787048770257007784027441750604
07248187680891577340092102561520453113150999104114621212581900017568894802351882575911344066176708098633881726008158810909280161896687717876573355039441 304
050653733908060445228661256369133625876946010840211356592001140126058451389149487289824809127857931621411793965707990335243651247702837646180202407966885441
8313186439722915366522444016609413981985082923759976192283747476075289221848248053581437798431754689620045252852975079968662316632150791193661637334
```

In the **cat** command of the above image, the space-separated integers represent the blocks of the signed and encrypted message.

Verifying signature and decrypting message generated in q2

File: q3.py and utils.py

Aim

To verify the signature, decrypt and print the message from q2 if the signature is valid.

Functions

1. *bin2str*: Given a binary string *s*, it converts the binary string to ASCII characters.
2. *unsign*: This is similar to encrypt function, it has been renamed for understandability.
3. *decrypt*: This function is used to decrypt the encrypted message *c* by performing the operation $c^d \equiv m \pmod{n}$ where *m* is the message, *d* is the secret exponent and *n* is the modulus.

Program

1. I read the private key of B and the public key of A since they are used in decrypting the message and decrypting the signature of the message respectively.
2. I also read the public key of B since A appended this to the message so that B can verify it.
3. I read the contents of the file *secret.txt* that was generated in q2, and it contains the signed and encrypted message.
4. Now, I try and decrypt the message and since q2 separated each block by a space, for each of those encrypted blocks,
 - a. I decrypt the block using the *decrypt* function mentioned above.
 - b. I run the encrypt process using the *unsign* method above, this helps in reversing the encryption done with the private key of A by A for signing the message.
 - c. Now, using the *bin2str* method mentioned above, I convert the block from the previous step into a string and append the contents final result variable.
 - d. If the decoding fails during conversion from binary to string, then either the signature or message was tampered and hence I print an error message and exit.
5. Now I have the message string and it contains two parts
 - a. The first part is the original message
 - b. The second part is the signature i.e., the public key of B in encrypted form

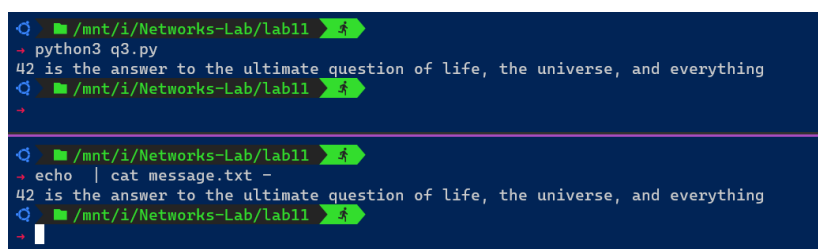
For example:

If message.txt contained the message "foo:bar", A encrypts and sends "foo:bar:(e, n_b)" now splitting the message by the last ":" will give us "foo:bar" and "(e, n_b)" i.e. the message and the public key.

So, now I split the string by the last ":" and the last part of string gives me the public key of B.

6. If the public key of B (which it knows) and the public key tuple obtained above are equal, then the signature is verified and hence I print the first part of the string obtained above. In case the signature is invalid, I print the error message and exit.

Figure 3: The output of q3 on successfully verifying the signature.



```
/mnt/i/Networks-Lab/lab11 ~$ python3 q3.py
42 is the answer to the ultimate question of life, the universe, and everything
/mnt/i/Networks-Lab/lab11 ~$
-
/mnt/i/Networks-Lab/lab11 ~$ echo | cat message.txt -
42 is the answer to the ultimate question of life, the universe, and everything
/mnt/i/Networks-Lab/lab11 ~$
-

```

Figure 4: Complete flow of the encryption and decryption process

```
/mnt/i/Networks-Lab/lab11 > python3 q1.py
Private and public keys for A successfully generated.
Private and public keys for B successfully generated.
/mnt/i/Networks-Lab/lab11 > python3 q2.py
Signed and encrypted message written to secret.txt
/mnt/i/Networks-Lab/lab11 > python3 q3.py
42 is the answer to the ultimate question of life, the universe, and everything
/mnt/i/Networks-Lab/lab11 >
```

Figure 5: Image showing the failure of signature verification.

<pre>/mnt/i/Networks-Lab/lab11 > python3 q3.py Message not verified! /mnt/i/Networks-Lab/lab11 ></pre>	<pre>/mnt/i/Networks-Lab/lab11 > sed '39q;d' q2.py msg += ":" + str((e, n_b)) /mnt/i/Networks-Lab/lab11 > sed '39q;d' q2.py msg += ":" + str((0, n_b)) /mnt/i/Networks-Lab/lab11 ></pre>
--	---

In the above image,

1. The first command in the right half shows the 39th line of the file q2.py. Here we can see that I am appending the public key of B to the message.
2. The second command shows the same line in q2.py but here I am appending a wrong public key (changed public exponent to 0) and hence when B tries to verify the signature, we should get an error.

As expected, in the left half, we get the error message "Message not verified!"