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 x q in the local variable n.
3. I compute the totient of n i.e., $\varphi(n) = (p - 1) \times (q - 1)$
4. Now I find the public exponent $(e: 5 < e < \varphi(n))$ 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 $e \equiv \varphi(n)$
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

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.

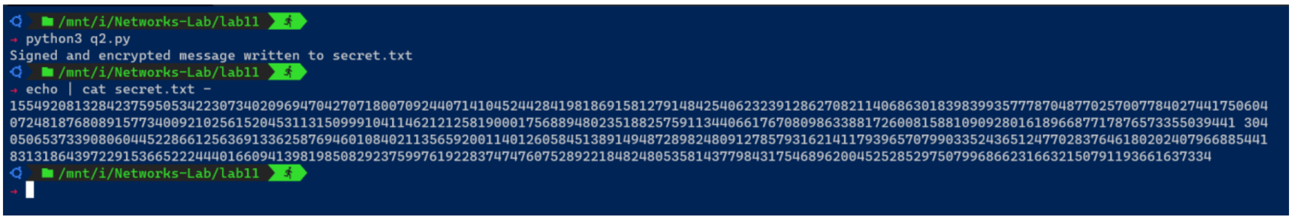
Functions

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



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 $c^d \equiv n$ 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.

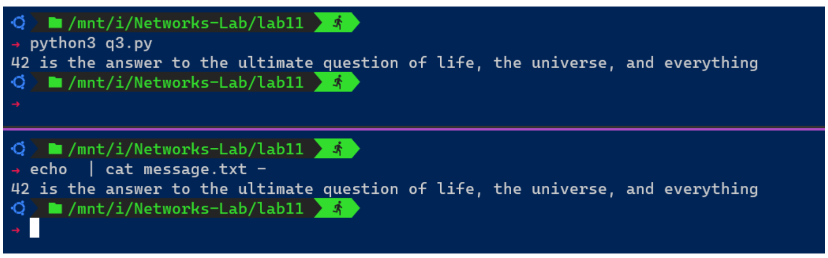


Figure 4: Complete flow of the encryption and decryption process

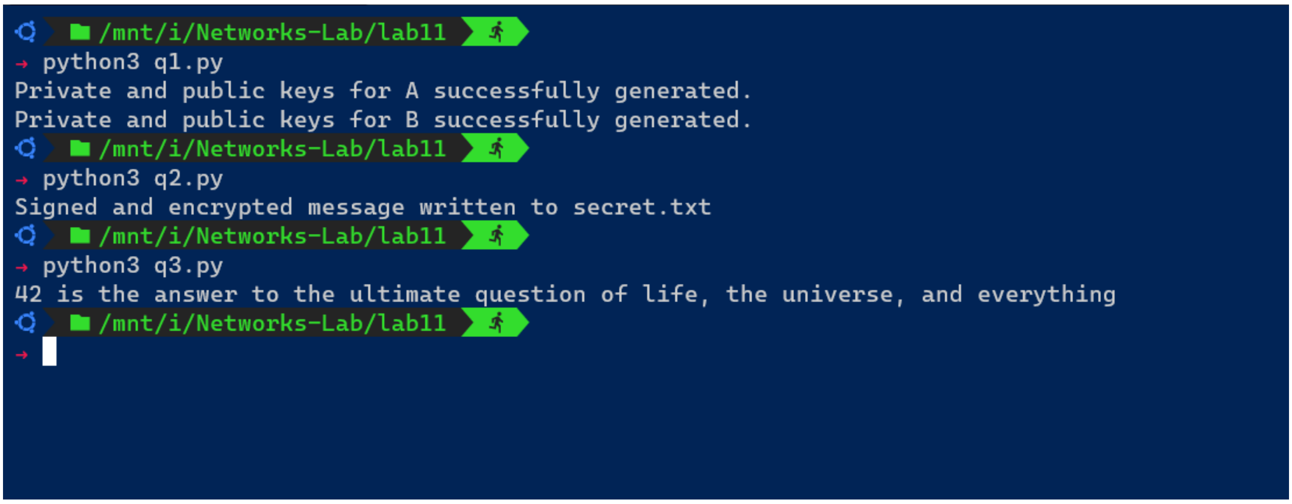
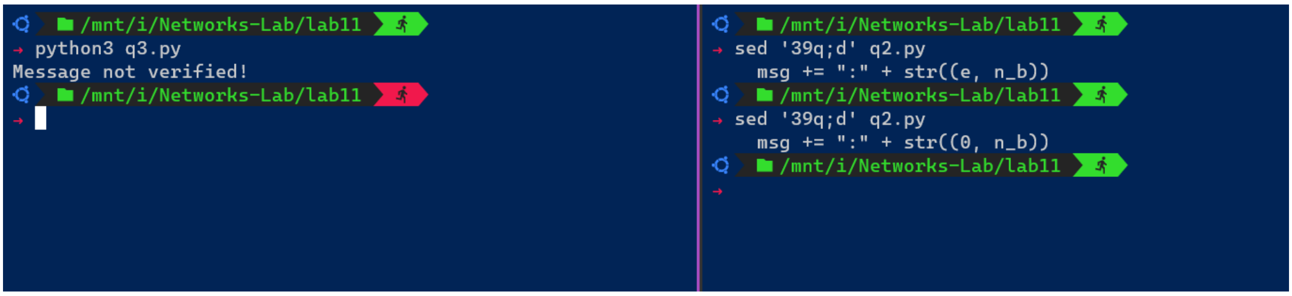


Figure 5: Image showing the failure of signature verification.



In the above image,

1. The first command in the right half shows the 39^{th} 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!*"