CS 4480: Computer Networks - Spring 2020

Programming Assignment: 3
Secure Messaging

There are **two** parts to this assignment:

PA 3 - A: Preliminary Design Document
PA 3 - Final: Complete Assignment

Due dates as indicated in Canvas

Assignment details

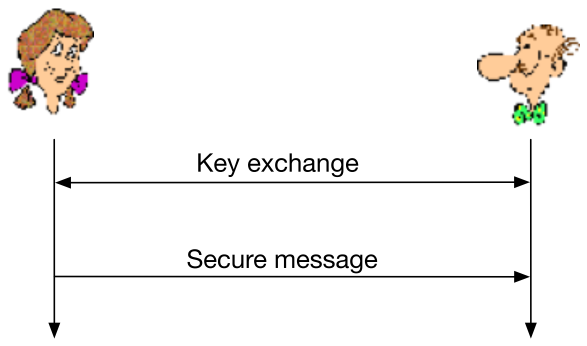

Figure 1: Overview of functionality

In this programming assignment,[1] you will make use of socket programming and crypto libraries to create a secure messaging transfer between a program running on Alice's host and a program running on Bob's host.

As shown in Figure 1, there are two stages associated with your Alice and Bob programs. The first part involves a *key exchange*: Alice sends a message to Bob asking for his public key. Bob responds with its public key that is digitally signed using a private key whose corresponding public key is known to everyone including Alice. Alice uses this well-known public key to obtain Bob's public key. (In a real situation, Bob will send a signed certificate that will include his public key but you do not need to deal with actual certificates in this programming assignment.) During the second part, Alice *securely sends a message* to Bob: Alice uses her private key, a symmetric key, and SHA-1, to implement the block diagram shown in Figure 2, for securely transmitting a text message (the message might be contained in a file). Bob implements the inverse of this block diagram to obtain the text message.

Table 1 describes the functionality that your programs should implement.

[1]Credit: This programming assignment is due to a similar assignment created by Sneha Kasera for his CS5480/6480 Computer Networks course.

Table 1: Programs representing Alice and Bob.

Alice:	**Bob:**
Request Bob's public key.	
	Provide Bob's public key together with a signed message digest of Bob's public key.
	Wait.
Obtain Bob's public key. Verify.	
Use a crypto library for integrity protection, encryption and signing of a text message to realize Figure 2.	
Transfer encrypted, integrity protected and signed message, together with symmetric key.	
	Received secure message from Alice. Use a crypto library to implement the inverse of Figure 2 to obtain Alice's message. Check for message integrity. Print the message
End.	End.

Before running the programs Alice generates a public and a private key (K_A^+) and K_A^-) and stores them. Bob also generates a public and a private key (K_B^+ and K_B^-) and stores them. Generate an additional public and private key pair (K_C^+ and K_C^-) such that Alice knows the public key, K_C^+, and Bob knows the private key (K_C^-). Bob uses the private key K_C^- to digitally sign the message digest of its own public key K_B^+. (In a real situation, a certificate authority will sign the message digest of Bob's public key but for this assignment you do not need to worry about a certificate authority. Instead, Bob does the job of the certificate authority.) Bob knows the public key of Alice, K_A^+. This means that Alice does not have to send a certificate, containing her key, to Bob. The top part of Figure 3 illustrates the key generation and distribution process.

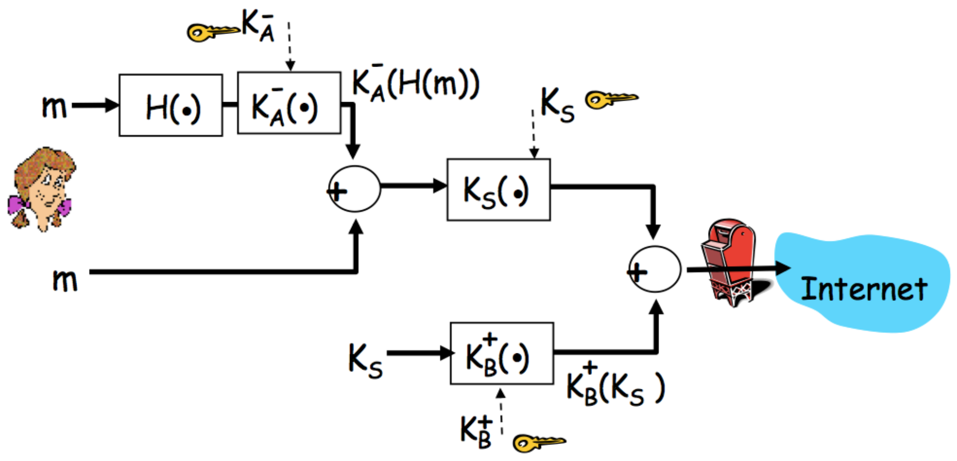

Figure 2: Sending side

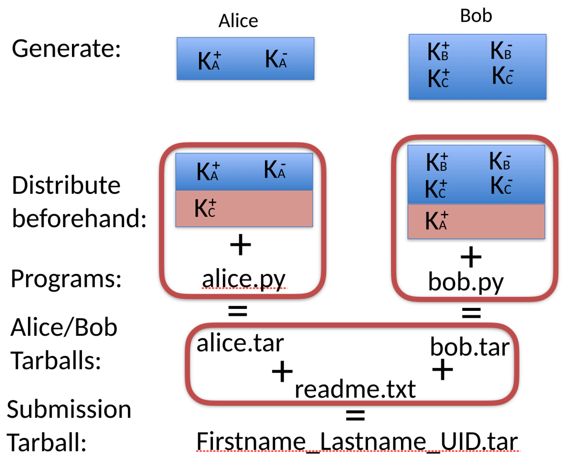

Figure 3: Key generation and packaging

Your assignment should be implemented in Python. You can use any version of Python that is available in the Cade Linux environment. Be sure to indicate in your `readme.txt` file which version you are using and how to execute your programs on Cade.

The main challenge in doing this assignment is to understand the crypto library used to implement the crypto functionality. We will make use of *openssl* and the Python PyCrypto [2] library. The benefit of *openssl* is that it can be used from the Linux commandline, which is very useful for understanding the functionality and for design purposes. You should make use of the man pages for *openssl* (and its sub commands) and the examples provided below. **To get full credit you should, however, make use of the PyCrypto library API rather than the commandline.**

In addition to understanding and properly using a crypto library, the different types of message components including message digest, symmetric key, encrypted message, etc should be properly delimited, e.g., by sending it in a file or in separate files, so that they could be properly parsed/separated at the other end. **This means that you will be designing a basic application level protocol to realize your secure messaging application.**

You should use RSA for public key encryption, 3DES for symmetric key cryptography and SHA1 for message digests.

Preparing your programs for grading Your assignment should be implemented using Python.

You should instrument your programs to be able to report progress to "stdout". A command line option, i.e., $-v$, should be provided to trigger this reporting when your programs are run. Your reporting should clearly show the progress for each interaction between Alice and Bob. You should also print all values (in hex format) for every step in the process to allow easy visual verification between programs. (E.g., print, for both Alice's and Bob's programs, the value of the message, of the keys, of the IV, of the digest etc.) **This will be the primary means by which your programs will be evaluated.**

You should package your code into two separate tar balls, alice.tar and bob.tar, which respectively contains the code and supporting files (i.e., public/private keys) in order to run the two programs on two separate Cade Linux machines to realize the interaction. This process is depicted in middle section of Figure 3.

[2]`https://www.dlitz.net/software/pycrypto/`

Grading and evaluation

What to hand in

PA 3 - A: Preliminary Design Document You need to submit via Canvas a preliminary design document. In this document you should show your design. At a minimum your design document will contain:

- A design of the application level protocol involved in realizing the two functional stages shown in Figure 1. I.e., message formats for the different stages.
- Screenshots of how you used the openssl commandline to "manually" design the proper interactions between Alice and Bob.
- A design of how these manual interactions will be realized when using the API of the PyCrypto library.

PA 3 - Final: Complete Assignment You submission should consist of a **single** tarball with the following naming convention:

```
Firstname_Lastname_UID.tar
e.g.,
Joe_Doe_u0000000.tar
```

You submission tarball file should contain the following:

1. The two programs and supporting material for your assignment. **(Packaged into two separate tar balls, alice.tar and bob.tar as described above and depicted in Figure 3.)**
2. A readme.txt file explaining how to run your program(s).

The complete packaging process is depicted in Figure 3.

Your submission tarball must be submitted on CADE machines using the handin command. To electronically submit files while logged in to a CADE machine, use:

```
% handin cs4480 assignment_name name_of_tarball_file
```

where `cs4480` is the name of the class account and `assignment_name` (pa1_a, pa1_final etc.) is the name of the appropriate subdirectory in the handin directory. Use pa3_final for this sub-assignment.

Grading

Criteria	Points
Preliminary design document	10
Program implemented according to specification and works correctly	80
Inline documentation & exception handling	10
Total	100

Note that if you choose to implement the assignment by using the *openssl* **commandline from within your program, the maximum points that you can get for the "Program implemented according to specification and works correctly" category will be 60.**

Other important points

- Every programming assignment of this course must be done individually by a student. No teaming or pairing is allowed.
- Your programs will be tested on CADE Lab Linux machines. You can develop your program(s) on any OS platform or machine but it is your responsibility to ensure that it runs on CADE Lab machines. You will not get any credit if the TA is unable to run your program(s).

OpenSSL Examples

(More detailed examples could be found in the book Network Security with OpenSSL by John Viega, Matt Messier & Pravir Chandra, O'Reilly 2002).

RSA commands:

```
openssl genrsa -out privatekey.pem 1024
```

Generates a 1024 bit RSA private key and writes it into the file privatekey.pem. The PEM format is widely used for storing keys, certificates etc.

```
openssl rsa -in privatekey.pem  -pubout  -out publickey.pem
```

Generates the corresponding RSA public key and writes it in publickey.pem.

```
openssl rsautl -encrypt -pubin -inkey publickey.pem -in x.txt -out y.bin
```

Contents of the file x.txt are encrypted and written to file y.bin using RSA public key from the file publickey.pem.

```
openssl rsautl -decrypt -inkey privatekey.pem -in y.bin -out x2.txt
```

Contents of file y.bin is decrypted to file x2.txt using the RSA private key from the file privatekey.pem.

Message Digest Commands:

```
openssl sha1 -sign privatekey.pem -out y2.bin x.txt
```

The SHA1 hash of the file named x.txt is signed using the RSA private key in the file rsaprivatekey.pem and the signature is written into the file y2.bin.

```
openssl sha1 -verify publickey.pem -signature y2.bin x.txt
```

The signature of the file x.txt that is contained in file y2.bin is verified using SHA1 message digest and the public key in file publickey.pem.

Symmetric Key Commands:

```
openssl enc -P -des3 -pass pass:cs4480
salt=92647A5B4984369E
key=3F6EB1A74175976B156334B8FACF49004DE14F51470BA29E
iv =34EF34F42D7535C7
```

Uses the password cs4480 to generate a symmetric key, salt and IV, and print those values (without doing any encryption).

```
openssl enc -des3 -in x.txt -out y3.bin -S 92647A5B4984369E
-K 3F6EB1A74175976B156334B8FACF49004DE14F51470BA29E -iv 34EF34F42D7535C7
```

Use the previously generated key, salt and IV to encrypt x.txt and outputs the cipher text in y3.bin

```
openssl enc -des3 -d -in y3.bin -out x3.txt -S 92647A5B4984369E
-K 3F6EB1A74175976B156334B8FACF49004DE14F51470BA29E -iv 34EF34F42D7535C7
```

Use the previously generated key, salt and IV to decrypt y3.bin and outputs the plain text in x3.txt

Other:

```
openssl base64 -e -in y2.bin -out y2.txt
```

Encode, using base64 encoding, the binary file y2.bin and outputs the encoded content into y2.txt

```
openssl base64 -d -in y2.txt -out y3.bin
```

Decode the base64 encoded file in y2.txt and output the resulting binary on y3.bin