CSCI 353: Programming Assignment 4

THIS PROJECT MUST BE DEVELOPED IN PYTHON 3.5+

(Python 2 submissions will receive a 0)

Due on November 13th, 11:59pm

Introduction

In this assignment you will implement secure communication between a client and a server that supports confidentiality, integrity and authentication. You will also implement a simple digital certificate management scheme.

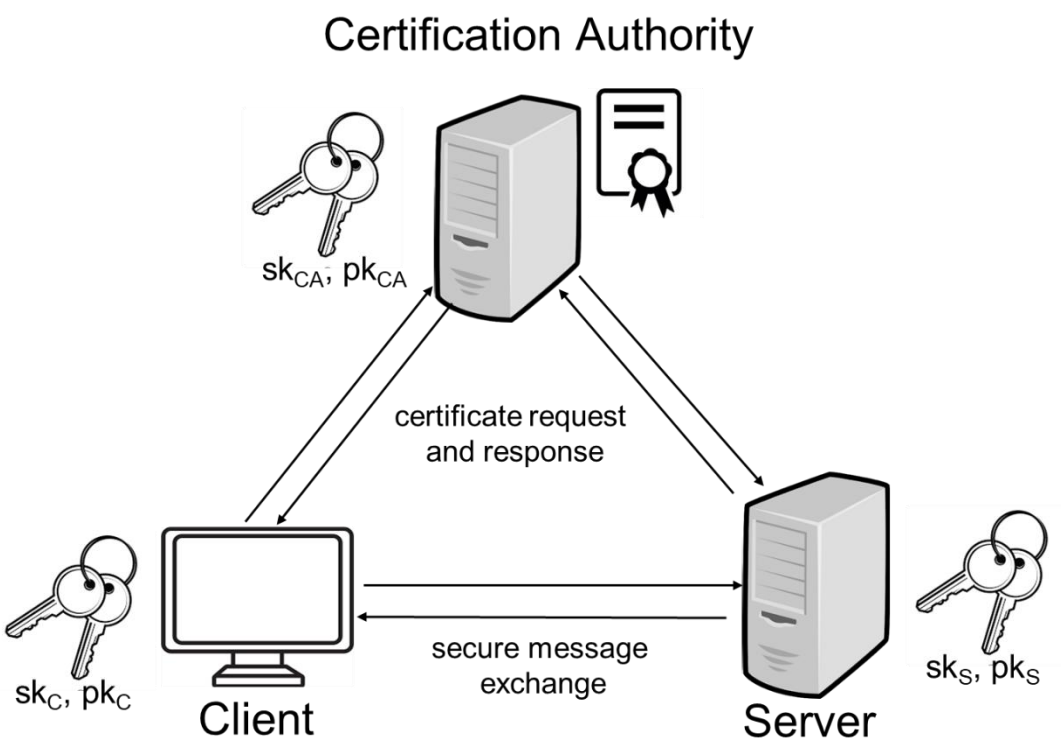
The main goal of this assignment is to give you hands-on experience in using some basic cryptographic concepts: RSA and AES encryption, SHA-256 hash, digital certificates and key exchange.

For this assignment you will use pyca/cryptography package https://cryptography.io/en/latest/ designed to expose cryptographic primitives and recipes to Python developers.

The package is divided into two levels:

1. Safe cryptographic recipes that require little to no configuration choices. These are safe and easy to use and don't require developers to make many decisions.
2. Low-level cryptographic primitives. These are often dangerous and can be used incorrectly. They require making decisions and having an in-depth knowledge of the cryptographic concepts at work. Because of the potential danger in working at this level, this is referred to as the "hazardous materials" or "hazmat" layer. These live in the cryptography.hazmat package, and their documentation will always contain an admonition at the top.

This site has very good documentation with examples. To figure out how to perform a certain task, just search in https://cryptography.io/en/latest/



You will implement three communicating entities that interact using the TCP socket connections: Client, Server and trusted Certification Authority. Before Client and Server can communicate securely, they need to acquire each other's public keys from Certification Authority. This requires registering a public key with Certification Authority.

To simplify this assignment, you will skip the public key verification step that uses X.509 certificates to validate that the key belongs to the right entity.

Once the public key is acquired, Client sends a signed, integrity and confidentiality protected message to Server. Server decrypts the message, checks the message integrity and responds to Client.

The detailed steps of this communication are described below. Your code should implement all the steps.

Certification Authority:

1. Generates RSA private and public key pair (sk_{CA}, pk_{CA})
   - Use public exponent = 65537, key size = 2048

https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/?highlight=rsa%20key%20generation#generation

2. Creates a self-signed certificate that is valid for 1 month
   - Use SHA-256 as a hash algorithm
   - Certificate attributes:

    - COUNTRY_NAME = US
    - PROVINCE_NAME = California
    - LOCALITY_NAME = Los Angeles
    - ORGANIZATION_NAME = USC
    - COMMON_NAME = Trusted CA
3. Gets certificate signing request from Server, extracts the attributes and generates a certificate C_S valid for 5 days
4. Gets certificate signing request from Client, extracts the attributes and generates a certificate C_C valid for 5 days
5. Sends the certificate C_C, CA's public key and Client's public key to Server
6. Sends the certificate C_S along with CA's public key and Server's public key to Client

Client:

1. Generates RSA private and public key pair (sk_C, pk_C)
    - Use public exponent = 65537, key size = 2048
2. Creates a certificate signing request and sends it to the Certification Authority
    - Certificate attributes:
        - COUNTRY_NAME = US
        - PROVINCE_NAME = California
        - LOCALITY_NAME = Los Angeles
        - ORGANIZATION_NAME = USC
        - COMMON_NAME = Client
3. Receives Server's certificate and Server's public key pk_S, from Certification Authority
4. Generates an AES-128 secret key K_S
5. Creates an example message m = "Cryptography is awesome!"
6. Creates a message signature σ by hashing the message m using SHA-256 and encrypting the result with its private key sk_C
    - Use PSS padding for signatures
7. Encrypts key K_S with Server's public key pk_S
    - Use OAEP paddings for encryption
8. Encrypts message m and the signature σ using key K_S
    - Use AES encryption with CTR encryption mode
9. Creates a new message m' by concatenating the results of the 3 previous steps (7, 8 and 9) and sends it to Server
10. Receives and decrypts a message from Server

Server:

1. Generates RSA private and public key pair (sk_S, pk_S)

- Use public exponent = 65537, key size = 2048

2. Creates a certificate signing request and sends it to Certification Authority
   - Certificate attributes:
     - COUNTRY_NAME = US
     - PROVINCE_NAME = California
     - LOCALITY_NAME = Los Angeles
     - ORGANIZATION_NAME = USC
     - COMMON_NAME = Server
3. Receives Client's certificate and Client's public key from Certification Authority
4. Uses its private key sk_S to decrypt and recover K_S
5. Uses K_S to decrypt the message to recover σ and m
   - Use CTR mode for AES decryption
6. Applies Client's public key pk_C to the signed message digest
   - Use PSS padding
7. Computes hash of the message m
8. Compares the result of step 6 with the hash computed in step 7
9. Creates an example new message m" = "I agree, cryptography is awesome!"
10. Encrypts the message with K_S and sends it to Client

Command Line Options:

> server –p portno -ss caIP –pp portno -m msg
where
-p portno the port number for Server
-ss <caIP> indicates the CA IP address
-pp <portno> port number for Server to connect to CA
-m msg message the server responds with

> client –s serverIP –p portno -ss caIP –pp portno -m msg
where
-s <serverIP> indicates the serverIP address
-p <portno> port number for Client to connect to Server
-ss <caIP> indicates the CA IP address
-pp <portno> port number for Client to connect to CA
-m msg message to be sent to the server

> ca –p portno
where
-p portno the port number for CA

If nothing is specified on the command line all programs should print the usage instructions and quit.

Required Terminal Output:
The following lines of output MUST be present in the terminal. You can also print additional debugging information in the terminal, but prepend debugging information with the keyword "DEBUG" so we can ignore it during grading.

The text in the angled brackets below should be replaced with the results from the communication between Client, Server and Certification Authority. Do not include the angled brackets in the output
Note: Output should match exactly.
Content of CA's certificate should be replaced with:

- COUNTRY_NAME = US
- PROVINCE_NAME = California
- LOCALITY_NAME = Los Angeles
- ORGANIZATION_NAME = USC
- COMMON_NAME = Trusted CA

CA Terminal

ca started on <1.2.3.4> at port <12345>…

ca public key: <content of CA's public key>

ca private key: <content of CA's private key>

ca certificate: <content of CA's certificate>

COUNTRY_NAME, PROVINCE_NAME, LOCALITY_NAME, ORGANIZATION_NAME, COMMON_NAME (Comma separated certificate)

received certificate request from <content of COMMON_NAME attribute> host <host name> port <port>

received certificate request from <content of COMMON_NAME attribute> host <host name> port <port>

sending certificate to <content of COMMON_NAME attribute>

sending certificate to <content of COMMON_NAME attribute>

Server Terminal

server started on <1.2.3.4> at port <12345>…

server public key: <content of Server's public key>

server private key: <content of Server's private key>

sending certificate request to CA: <host> port <port>

received client certificate: <content of Client's certificate>

COUNTRY_NAME, PROVINCE_NAME, LOCALITY_NAME, ORGANIZATION_NAME, COMMON_NAME (Comma separated certificate)

received message from client: <content of Client's decrypted message>

integrity check: calculated = <content of calculated hash> received = <content of decrypted hash>

sending message to client: <content of encrypted Server's message>

Client Terminal

client started on <1.2.3.4> at port <12345>…

client public key: <content of Client's public key>

client private key: <content of Client's private key>

sending certificate request to CA: <host> port <port>

received server certificate: <content of Server's certificate>

COUNTRY_NAME, PROVINCE_NAME, LOCALITY_NAME, ORGANIZATION_NAME, COMMON_NAME (Comma separated certificate)

generated AES key: <content of the secret key>

message signature: <content of the signature>

encrypted message: <content of encrypted message>

sending encrypted message to server

received server response: <content of decrypted Server's message>

Code and Collaboration Policy

You can discuss the assignment and coding strategies with your classmates. However, your solution must be coded and written by yourself. Please refer to the plagiarism policy in the course syllabus.

The submissions will be run through code similarity tests. Any flagged submissions will result in a failing score. Keeping your code private is your responsibility.

Submission Instructions

You can develop and test your code on your own machines. Create a compressed tar file which includes a README and the source code.

To submit, create a folder called LASTNAME_FIRSTNAME with the above files. Tar the folder LASTNAME_FIRSTNAME. Submit the tar file on blackboard.

The README must contain: your USC ID, compiling instructions, additional notes on usage if needed. (e.g., The default IP address used for grading is 127.0.0.1[localhost]. If you wish to use any other addresses, please specify)

Make sure you add the directives to support direct execution. The directory structure should look like this:

LASTNAME_FIRSTNAME

- client.py
- server.py
- ca.py

->README.txt

We will then run your programs using a suite of test inputs. After running the program, we will grade your work based on the **Terminal output**. It is recommended that your implementation be modular. This means that you should follow good programming practices—keep functions relatively short, use descriptive variable names.