CSE 350/550: Assignment4

Project 1: Digitally Signed Degree Certificates

1. **RSA Algorithm implementation**

#### RSA(M, key)

M is a positive integer and the key is tuple (e,n). Returns M^(e) mod n . We have implemented a fast algorithm for computing modulo exponent in log2 e iterations.

#### RSA\_encrypt\_string(msg, pk)

msg is a string of characters to be encrypted. pk is key in the form of (e,n). First, characters are converted to bytes and then each byte is encrypted separately using RSA().The numbers obtained from RSA() are concatenated to a long string separated by , . The final comma separated string is returned.

#### RSA\_decrypt\_string(msg, pu)

msg is a string of numbers separated by commas. pu is key in the form of (d,n). The string is split on commas to get individual numbers. Each number is separately decrypted by calling RSA(). Numbers from RSA() calls are converted to equivalent characters, appended to a string and the string is returned.

#### RSA\_encrypt\_bytes(msg, pk)

This function encrypts each byte of the input bytestring msg separately to output a list of numbers, 1 for each byte. Eventually a string of numbers separated by comma is returned.

#### RSA\_decrypt\_bytes(msg, pu)

It takes in as input a string of numbers separated by commas, removes commas and makes a list of numbers. It then decrypts each number separately to get ASCII codes, converts each ASCII code to byte using the to\_bytes function in the little endian mode. The end result is a bytestring.

#### RSA\_keygen(p,q)

Given two primes p and q. Returns e,d,n where n = p\*q.

phi = (p-1)\*(q-1). e is any number in the range of 2 to phi such that gcd(e,phi) = 1. d is any number in range of 2 to phi such that (e\*d) mod phi = 1.

1. **Certification Authority (CA) implementation**

#### handle\_client(self, connection)

This function, which is called upon getting a new client connection request, takes in the socket created by socket.accept() method as input and basically supplies the certificate to the client. It does this by first sending an introductory message confirming the success of the connection, then it receives the ID for the public key certificate which is desired by the client. It supplies this ID to the get\_certificate function, which ends up returning the encrypted certificate, which is then sent to the client and the connection is closed.

#### add\_publickey(self, id\_client, key)

This function adds a key for a given client\_id to the keystore of the CA.

#### get\_certificate(self, client)

This function returns the encrypted certificate which has the Public Key (PU) of the client. It takes in the client ID as input and retrieves its public key from its keystore and stores it in the PUA variable. The current timestamp is recorded in the TA variable, and the duration of validity is recorded in the DURA variable (it has been assumed to be 5 years here). The certificate is finally of the format client\_id :: PUA :: TA :: DURA :: CA\_id. This certificate string created is encrypted using the private key of the CA(as the purpose is authentication of the certificate) and returned.

#### Class data fields

The CA class has a map\_pukeys attribute that stores the public keys for the respective client IDs. The self.privatekey attribute stores its private key and self.id stores the CA's id which is assigned after being auto incremented upon initialisation of every new CA object.

1. **Client Functions**

#### request\_ca(id)

This function opens a socket connection with the CA at port 8765 (the CA port) and sends the input argument “id” to the CA and returns the public key certificate for this ID which was returned by the CA, finally closing the socket connection.

#### getkey\_from\_certificate(cert)

This function takes in as input the certificate provided by the CA, and retrieves the public key from the certificate, by first decrypting the certificate using the ca’s public key, turn isolating the fields of the certificate using the “::” delimiter, the second of the certificate is the key string of the form (e,n). It is further split at “,”. The split strings are converted to integers to form e and n and returned as a tuple.

1. **Server Functions**

#### add\_watermark(filename,timestamp)

We use the PyMuPDF python library (imported as fitz here) for adding a watermark. This function takes in as input the filename which has the path of the file to be watermarked and the current timestamp. The text to be watermarked is "IIIT DELHI ACADEMICS: "+timestamp.This is done using the TextWriter object of the fitz library using the append and write\_text methods. The watermarked document is saved with the suffix \_stamped appended to the original filename and the new file location thus created is returned .We assume that it is a single page file here, and thus watermark on only the first page. The functionality can be easily extended to supported multi-page documents.

Note: the request\_ca(id) and getkey\_from\_certificate(cert) functions have been implemented in the exact same way as they were implemented for the client, so their explanation has been skipped.

#### gettime()

Makes a request to NTP server, with address at uk.pool.ntp.org, fetches current GMT time and returns a string representation of that time

1. **entity implementation**

#### Class datafields

name: a string

PR: the private key of the entity, in the form of a tuple of two integers.

Registrar and Director would be objects of entity class.

#### add\_sign(msg0)

msg0 is a byte string. Entity adds its name and time of signing at the end of msg0. Time of signing is obtained from call to gettime()

Effectively returns msg1, where

msg1 = msg0 || name || timeofsigning || s3 || s4

s3 = 4 bytes representing length of name

s4 = 4 bytes representing length of timeofsigning

#### add\_hash(msg1)

msg1 is a byte string. Computes sha256 hash of msg1, and encrypts it using private key PR. Effectively returns msg2, where

msg2 = msg1 || encrypted hash || s5

s5 = 4 bytes representing length of encrypted hash

1. **Program Flow**

**Starting of CA**

CA generates and stores its private key. Then, CA stores the public keys of the Director, Registrar Institute Server( from here on will be called server), and all the students. Then CA binds to IP:127.0.0.1 , port:8765 and starts listening for incoming connections. Whenever it receives a connection , it starts a new thread which calls handle\_client().

**Starting of server**

First, we create a memory map DB with keys = (name, rollno) and values = (path to degree pdf, path to grades pdf). Then the server binds to 127.0.0.1, 12345 and starts listening for connections. For each connection, it starts a new thread which calls handle\_client().

**Starting of client**

Client requests public key certificate of the server from CA and extracts public key from it. Then the client calls request() with parameters : name, rollno, private key of user, etc.

Then, the client creates a socket to connect to the server. Then the client creates a message= name,rollno and computes the sha256 hash of the message. Hash is concatenated to the end of the message. Final message is encrypted using the public key of the server and sent to it.

Server decrypts the message using its private key. If name, rollno is not in DB, server responds NAME AND ROLLNO NOT FOUND. Then, the server checks if the message has been tampered with by comparing its computed hash with the attached hash. If tampered, the server responds INTEGRITY FAILURE. Otherwise, the server responds SUCCESS. Server requests certificate of specified rollno from CA to get student’s public key. Then, the server reads the PDFs in byte format & add watermark to each of them. Then, PDF1 and PDF2 are concatenated to get combined bytes PDFs as follows.

msg0 = PDFs || s1 || s2

s1=length of PDF1

s2=length of PDF2

Then, the Registrar digitally signs the PDFs as follows

msg1 = registrar.add\_sign(msg0)

msg2 = registrar.add\_hash(msg1)

Similarly, then the Director digitally signs msg2 as follows.

msg3 = director.add\_sign(msg2)

msg4 = director.add\_hash(msg3)

**Overall scheme:**

msg1 = msg0 || "Registrar" || time\_of\_signing || s3 || s4

s3=size of "Registrar"

s4=size of time\_of\_signing

msg2 = msg1 || E(hash of (msg1), key = PR\_Registrar) || s5

s5=size of encryptedhash

msg3 = msg2 || "Director" || time\_of\_signing || s6 || s7

s6=size of "Director"

s7=size of time\_of\_signing

msg4 = msg3 || E(hash of (msg3), key = PR\_Director) || s8

s8=size of encryptedhash

Final msg = E(msg4, key = PR\_user)

For confidentiality purposes, we encrypt the final message with the public key of specified roll no. Then, the server sends message size (encrypted) to the client.

Client decrypts received message size using its private key. Client then reads the complete message sent from the server. Client decrypts it using its private key.

To authenticate the Director, Client then requests a certificate of Director from CA. Client then extracts msg3 from the received message and computes its sha256 hash. Client also decrypts hash of msg3 by using Director’s public key.

Client then checks whether both hashes match or not.

To authenticate the Registrar, Client then requests the certificate of Registrar from CA. Client then extracts msg1 from the received message and computes its sha256 hash. Client also decrypts hash of msg1 by using Registrar’s public key. Client then checks whether both hashes match or not.

Client then extracts msg0 from msg1. Client uses s1, s0 from msg0 to extract PDF1 and PDF2 from msg0. Client then saves both the PDFs and closes the connection.

## 

## Issues Addressed

1. How and where do you get the correct GMT date and time? Is the source reliable and the GMT date and time obtained in a secure manner?

We have used an NTP server to retrieve the most accurate GMT time, in particular the uk.pool.ntp.org NTP server. This has been done using the ntplib library of python. NTP uses the intersection algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency, using high-precision timekeeping devices such as atomic clocks at the lowest layers of the hierarchy. NTP can usually maintain time to within tens of milliseconds over the public Internet and is thus considered to be extremely accurate and reliable, and used globally.

1. How do you ensure that only the graduate is able to download it (by providing information beyond the roll no, such as date of birth, home pin code, etc.)?

We have used the public key infrastructure for authenticating the client and communicating the messages. The degree certificates sent are encrypted using the public key of the student in question(which in turn is securely obtained from the certificate authority), and thus only the student themselves can decrypt the message to obtain their gradesheets and degree.

1. Should the graduate decide to share the document with others, how can one trace the origin of the document (could watermarks be useful?)?

To address this issue, we have added watermarks to both the degree certificates as well as the grade sheets. For the purpose of demonstration, we have used an arbitrary text “IIIT DELHI ACADEMICS:<current timestamp>” as a watermark, but it can be standardized as per the decision of the University Authorities.

1. Do we need to have access to public-keys, and if so how?

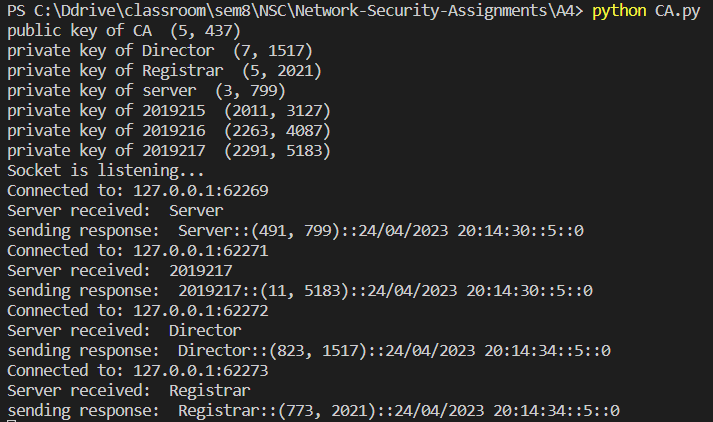
Yes, Client needs to have access to the public key of the Server, Director, and Registrar. Server needs to have access to the public key of the requesting student. For this we have implemented Certification Authority. Any entity can request a public key certificate of any other entity from CA. The received certificate is decrypted using the public key of Certification Authority.

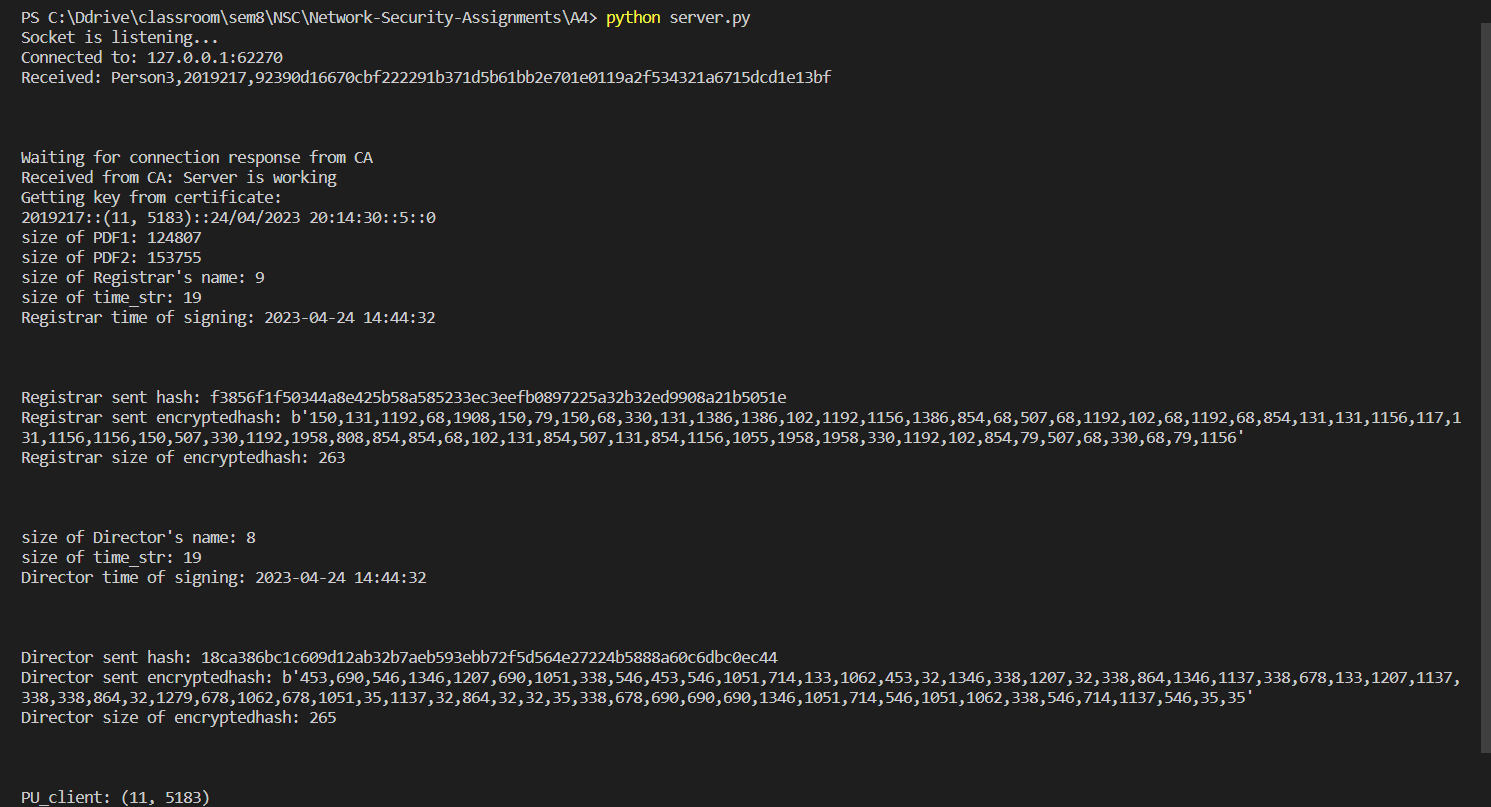
Bonus: How do you get the document to be digitally signed by two persons (say the Registrar and the Director?

Here, we have added 2 signatures to our message containing the grade sheet and the degree certificate of the student.

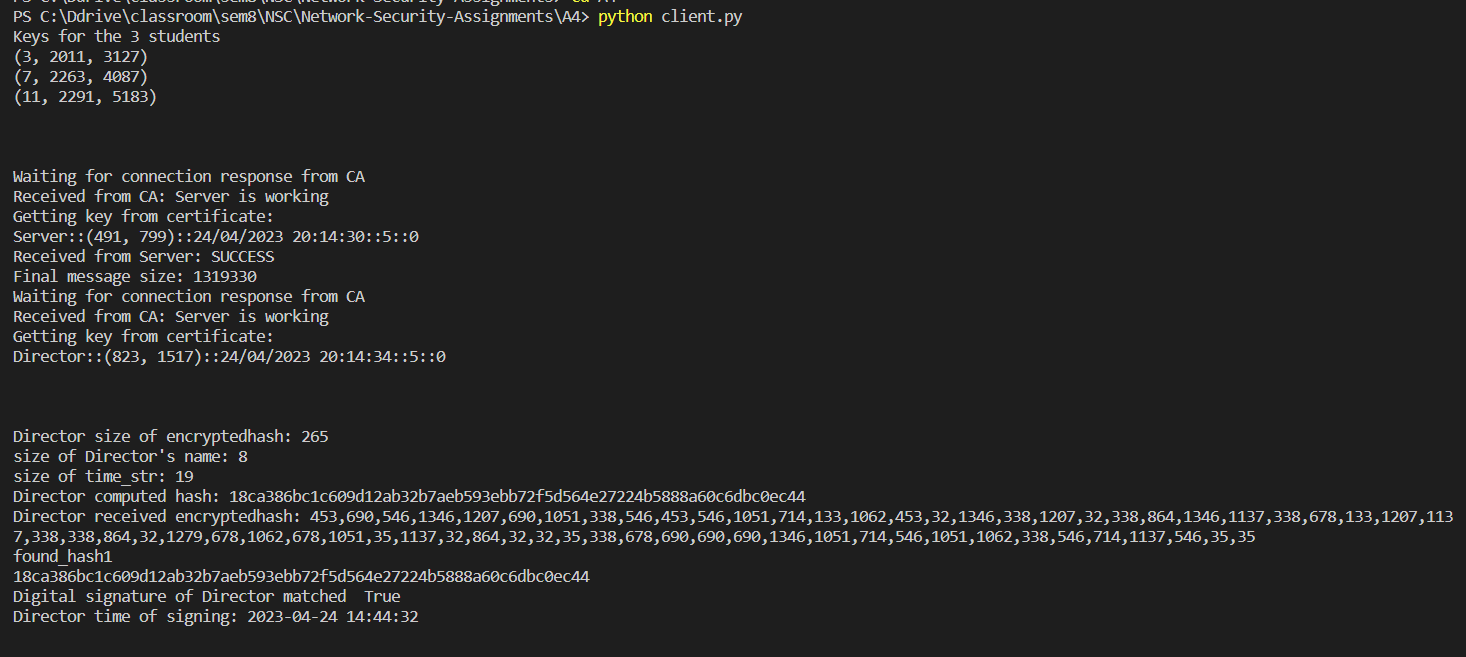
The first signature is the hash of ( pdfs || name of Registrar || time of signing) encrypted using the private key of the Registrar. Then the signature is appended to the input message to get msg2. The second signature is the hash of (msg2 || name of Director || time of signing) encrypted using the private key of the Director. So, we implemented a chain of signatures such that the Registrar signs before the Director.

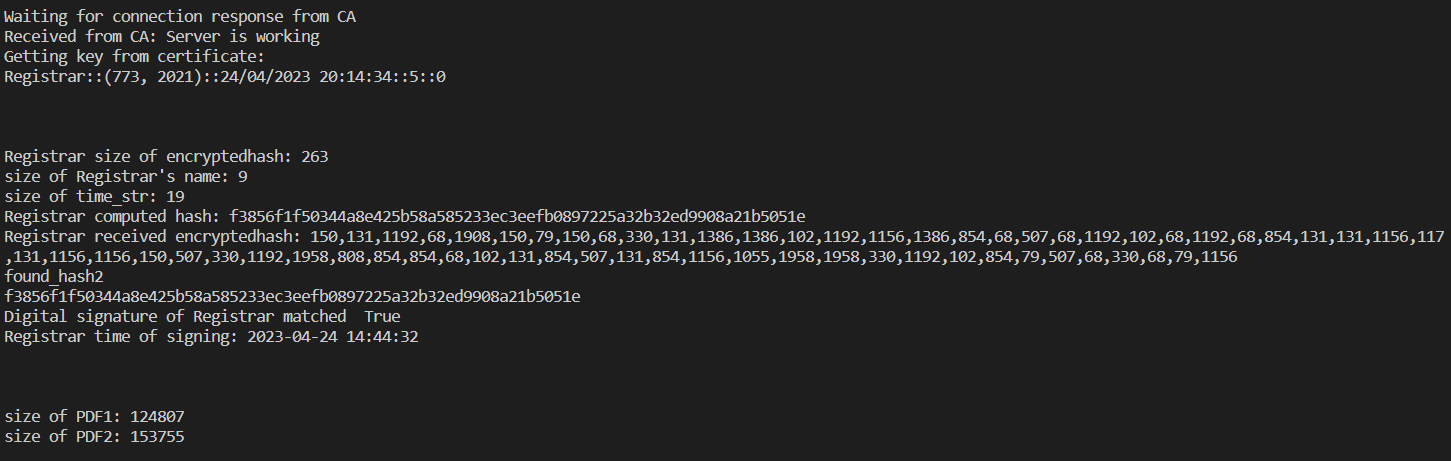
**Code Output:**

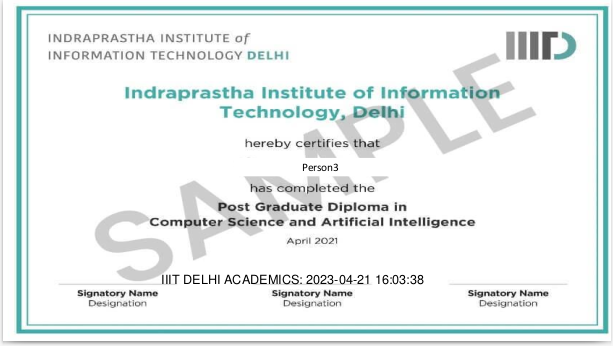
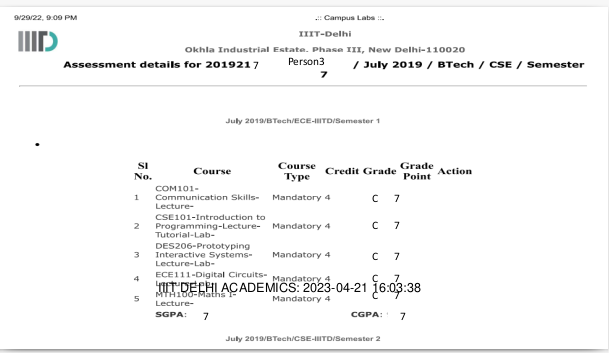
****

****

****

****

****

****