CO7099 Assignment 01

Released Jan 30, 2025

Deadline Friday Feb 21, 2025 12:00 noon

AI policy: Generative AI must **NOT** be used when attempting any part of this assignment.

This assignment is **worth 20% of the module mark**.

It assesses the following learning outcome in the module specifications: "Design and implement secure network applications using standard cryptographic libraries". Specifically, it assesses your knowledge in using the Java cryptographic API and related features, and your understanding of cryptographic concepts such as public key cryptography. **You will need to know the topics from the first 4 weeks (up to authentication) in lectures and labs to complete this assessment**.

You can form groups of up to 2 students for this assignment. Instructions will be provided in a separate file on blackboard that will be visible under Assessment and Feedback -> Programming Assignment.

Clarifications and amendments may be announced on Blackboard for this assignment.

Objective: You will write a Java client-server system that allows users to download files from the server with end-to-end encryption and authentication.

1. Client-server architecture, public and private keys

- The system consists of a client and a server Java program, and they must be named **`Client.java`** and **`Server.java`** respectively. They are started by running the commands:

  ```
  java Server port
  java Client host port userid
  ```

  specifying the hostname and port number of the server, and the userid of the client.
- The server program is always running once started and listens for incoming connections at the port specified. When a client is connected, the server handles the request, then waits for the next request (i.e., **the server never terminates**). For simplicity, you can **assume that only one client will connect to the server at any one time**.
- Each user has a unique userid, which is a simple string like alice, bob etc. The server has userid "server". Each user is associated with a pair of RSA public and private keys, with filenames that have .pub or. prv after the userid, respectively. Thus, the key files are named `alice.pub`, `server.prv`, etc. These keys are generated separately by a program RSAKeyGen.java. More details are in the comments of that program. (This is the same file that was provided for Lab 3)
- **It is assumed that the server already has the public keys of all legitimate users**, and each client program user already has their own private key as well as the public key of the server. They obtained these keys via some separate mechanism not described here, prior to the execution of the client and server programs, and is not part of these programs. **The client and server programs never create any new keys or distribute them**.

- All the key files are in the same folder where the respective client/server program runs from. They must not be read from other folders. **Your programs must not require keys that they are not supposed to have.**

2. The authentication and key agreement stage

- When the client program starts, it connects to the server and sends it the userid of the client and 16 fresh random bytes, both encrypted with **RSA/ECB/PKCS1Padding** and with an appropriate RSA key so that only the server can decrypt it. It also generates a signature of these encrypted bytes using the **SHA1withRSA signature algorithm**, with the appropriate RSA key that proves the identity of this client. (RSA keys can also be used for signatures; in this system the same set of RSA keys are used for both encryption and signature purposes.) **The encrypted contents and the signature are sent to the server.**
- Upon accepting the connection of a client and receiving the contents in the previous step, the server program decrypts them (using an appropriate RSA key) to obtain the client userid and those 16 bytes. It also verifies the signature (using an appropriate RSA key).
- Next, the server generates its own 16 fresh random bytes, concatenates it after the client's 16 bytes, and encrypts the combined 32 bytes with **RSA/ECB/PKCS1Padding** and with a key so that only the corresponding client can decrypt it. It also generates a signature of the encrypted bytes using the **SHA1withRSA algorithm** and with a key that proves the identity of the server. **The encrypted key and the signature are sent to the client.**
- The client receives these contents, verifies the signature (using an appropriate RSA key), and decrypts the encrypted bytes (using an appropriate RSA key) to obtain the 32 bytes. It checks whether the first 16 bytes are indeed those sent by the client itself initially; if not, it should terminate the connection (and do not continue with the rest of this protocol).

- Both the client and server (independently) **use these 32 bytes to generate a 256-bit AES key**. **Note that these unencrypted bytes (and the resulting key) must never be sent across the network.**

3. The file transmission stage

- From now on, the client and server will take turns to send messages to each other: client to send requests, and the server to send responses. Unless otherwise specified, all these messages should be encrypted by the sender before sending, with **AES/CBC/PKCS5Padding and the AES key agreed** in the previous stage, and decrypted by the recipient upon receipt.
- The CBC mode needs a 16-byte initialisation vector (IV); here we will hash the 32 bytes in the key agreement stage, using MD5, and take the resulting 16 bytes as the IV for the first message. Each subsequent message will use the MD5 hash of the previous IV as the new IV.
- The client program repeatedly prompts the user to type in what they want to do. The **user can type in one of three commands**: "ls", "get filename", or "bye". The client program should send the command and the name of the requested file (if any) to the server. At least the filename should be encrypted; you can choose to encrypt the command itself or not. The server will react accordingly, as specified below.
    1. **If the command is "ls"** (same as the linux command with same name), it is a request for the server to send a list of filenames of all files available for download. The server should send back the names of all files in its current working folder. But since the key files are also in the folder, it should filter out all ".prv" files and do not include any of them in the list. (In the lab class we will give more information on how to do this.)
    2. **If the command is "get filename"**, it is a request for the server to send the contents of the file of that name. It reads and encrypts the content of that file and sends it to the client. The

client decrypts the received content and saves it locally, with the same filename, in the current working folder. The requested file should be from the server's current folder, excluding any .prv files (i.e., what the client would see if they use the ls command). If it is not one of those files, it should notify the client that the file does not exist. (The lab class will also talk more about such if/else situations.)

3. **If the command is "bye"**, it means the client has no further requests. Both sides should end the connection. The client simply quits, whereas the server waits for the next client.

4. Program outputs

- The client program should prompt for the three commands and display any responses appropriately. There is **no fixed format** for how these screen outputs should look like. **It should not display the contents of the file on screen.**
- The server program **should print on the screen the (plaintext) userid of each client upon connection**, any **commands received (in plaintext)**, and **any issues encountered such as failed signature verification**. Again, there is no fixed format required for how these printouts should look like.
- A demo program will be shown in class and in a recorded video, but you are not required to follow its exact input/output.

5 Error handling

- If at any point in the protocol, some cryptographic checks went wrong, such as failed signature verification or failed decryption, the program should terminate the connection (and do not continue with the rest of the protocol).
- Your program is not required to handle errors other than those specified above. For example, you can assume the following won't happen (or in other words your program can behave arbitrarily if they do happen):

a) No user of that name exists (no such key files exist)
b) A file of the same name already exists on the client side (you can quietly overwrite the existing file)
c) The client command is not one of the three stated above (though it would be nice if the client program informs the user that it is an unrecognised command and prompt them to enter again)

- When one side terminates the connection (e.g., because of failed signature), the other side is allowed to behave arbitrarily including crash (though it would be nice if the server doesn't crash in such circumstances, but stay alive and wait for the next client.)

Marking criteria: Provided in a separate file on blackboard that will be visible under Assessment and Feedback -> Programming Assignment.

Submission instructions

- Submit your completed work on Blackboard ("Assessment and Feedback", then "Programming Assignment").
- Please also see the document on important instructions for group work.
- Your submission should consist of the two files `Client.java` and `Server.java`. **They must be of these exact names (with the exact upper/lowercase)**. Should you wish to, you can include additional java files. But **all of them must be .java files** and must be uploaded as separate files (**not zipped**: you can select and upload multiple files in a single submission in Blackboard).
- **AI policy:** Generative AI must **NOT** be used when attempting any part of this assignment.
- Plagiarism will be treated strictly according to standard university and departmental procedures. Your submissions will be sent to a plagiarism detection service (MOSS).

- In line with university policy, marking will be done anonymously. Only the Blackboard-supplied userid / student number will be visible in marking.
- For the above two reasons, do not include your name, userid, student number, or any other personally identifiable information in your programs.