# CS432/532 Computer and Network Security
# Spring 2021 - Term Project

*Secure File Relaying Application with Limited Server Trust*

Project Step 1 Due: April 27, 2021, Tuesday, 21:00 (to be submitted to SUCourse+)
Demos: Time and Schedule will be announced later

Project Step 2 Due: May 20, 2021, Thursday, 21:00 (to be submitted to SUCourse+)
Demos: Time and Schedule will be announced later

## Submission and Rules

- **CS532 students** should complete the project alone. CS532 students have an option to propose their own projects instead of doing this one; however, in such a case, the project proposal must be sent to the instructor before the deadline of the first step of the project. If no proposals are received, we assume that you will do this project.
- **CS432 students** can work in groups of 4-5 people in both steps of the project, not less not more. You have to group yourselves until the April 22, 2021. You will be informed by your TA Kerem Örs about how to send the group information.
- Equal distribution of the work among the group members is essential.
- All group members should appear at the demos.
- All group members should submit the complete project in each step to SUCourse+. No email submissions please. Make sure that all group members submit the same version. If a group member does not submit, we will assume that this person has been kicked out of the group and he/she will receive zero.
- The submitted code will be used during the demo. No modification on the submitted code will be allowed.
- Submission steps:
    - Delete the content of debug/release folders in your project directory before submission.
    - Create a folder named ***Server*** and put your file server related codes here.
    - Create a folder named ***Client*** and put your client related codes here.
    - Create a folder named ***XXXXX_Surname_Name***, where *XXXXX* is your SUNet ID (e.g. *fkerem_Ors_FaikKerem*). Put your *Server* and *Client* folders.
    - Compress your *XXXXX_Lastname_Name* folder **using ZIP**. Please **do not use** other compression utilities like RAR, 7z, ICE, bz2, etc.
- You will be invited for a demonstration of your work. Date and other details about the demo will be announced later.
- Late submission is allowed only for one day (24 hours) with the cost of 10 points (out of 100).

## Programming Rules

- Preferred languages are C#, Java and Python, but C# is recommended.
- Your application should have a graphical user interface (GUI). **It is not a console application!**
- Your code should be clearly commented. This may affect up to 10% of your grade.
- Your program should be portable. It should not require any dependencies specific to your computer. We will download, compile and run it. If it does not run, it means that your program is not running. So do test your program before submission.
- In your application when a window is closed, **all threads related to this window should be terminated.**

We encourage the use of course's WhatsApp group for the general questions related to the project and for finding group members. For personal questions and support, you can send email to course TAs.

Good luck!

Albert Levi
Faik Kerem Örs - fkerem@sabanciuniv.edu
Mehmed Yuşa Ergüven - merguven@sabanciuniv.edu

## Introduction

In this project, you will implement a client-server application for establishing a *secure* and *authenticated* file relaying mechanism. In the application that you will implement, there are *Client* and *Server* modules.

The *Server* module:

- authenticates the clients,
- verifies the authenticity of the files uploaded by the clients,
- receives and stores the files in an encrypted form,
- distributes requested files to users in a secure and authenticated way.

The *Client* module:

- gets authenticated to the server,
- connects to the server to upload an encrypted file in a secure and authenticated way.
- requests files from server and download in a secure and authenticated way.

Roughly, client authentication to the server, starting up the server and security configurations (loading the keys) are the first step of the project. The rest is the second step.

You should design and implement a user-friendly and functional graphical user interface (GUI) for client and server programs. In both of the steps, all activities and data generated by server and client should be reported in text fields at their GUIs. These include (but not limited to):

- RSA public and private keys in hexadecimal format
- AES keys and IVs in hexadecimal format
- Random challenges and responses calculated for the challenge-response protocol runs in hexadecimal format
- Digital signatures in hexadecimal format
- List of online clients (on the authentication server side)
- Disconnected or newcomer clients' usernames (on the server side)
- Verification results (digital signatures, HMAC verifications)
- File transfer operations and file details
- File access requests and permissions, permission outcomes
- HMAC values, session keys, etc. in hexadecimal format

## Project Step 1 (Due: April 27, 2021, Tuesday, 21:00)

In the first step of the project, the client performs *connection*, *user authentication* and *disconnection* operations. You will implement an authentication protocol between client and server. Server authenticates each user (client) if he/she is a known and valid person using a challenge-response protocol. The details are explained in the following subsections.

**Connection Phase**

Please remark that this is a client-server application, meaning that the server listens on a predefined port and accepts incoming client connections. The listening port number of the server must be entered via the server GUI. Clients connect to the server on the corresponding port and identify themselves with usernames. Server needs to keep the usernames of currently connected clients in order to avoid the same username to be connected more than once at a given time. There might be one or more different clients connected to the server at the same time. Each client knows the IP address and the listening port of the server (to be entered through the GUI).

**Authentication (Secure Login) Phase**

In this project, the usernames, passwords, public and private keys of each user are given to you (see below "Provided RSA Keys and Other Secret Info" section and the project pack). The RSA-4096 private key of each user is given in encrypted form. The encryption is done using AES-256 in CFB mode. For the key and IV of this encryption, SHA-384 hash of the password is used, first 32 bytes of the hash output, i.e. the byte array indices [0…31], being the key and last 16 bytes of the hash output, i.e. the byte array indices [32…47], being the IV.

This phase is the implementation of a secure login protocol for the client to get authenticated to the server. In this phase, a challenge-response protocol is run between client and the server. After client connects to the server and identifies herself/himself with a unique username, she/he enters password from the client GUI. The encrypted 4096-bit RSA private key, which is given in the project pack, is decrypted using the hash of the entered password; this decryption operation will use the same cryptographic algorithms and parameters mentioned above. After this decryption operation, if the password is entered correctly, we obtain the decrypted RSA private key. You should not store this decrypted private key in any file; just keep it in memory during a session. If the password is entered wrong, the decryption operation would fail (probably by throwing an exception) and you'd understand that you entered the password wrong. In such a case, you have to inform the user about the wrong password and ask for it again. Depending on the platform and the programming language you are using, the decryption operation of the encrypted private key may not fail even a wrong password is entered. If this is the case, you have to find ways to understand the wrong password entry within the authentication protocol. However, <u>you can never store the password in any form to understand its correctness</u>.

Meanwhile, the server sends a 128-bit random number to the client to initiate the challenge-response protocol. After the decryption of the private key, the client signs this random number using his/her private RSA key and sends this signature to the server. The hash algorithm used in signature is SHA-512. If the server cannot verify the signature, it sends a negative acknowledgment message to the client and closes the communication. If the server can verify the signature, the client is authenticated. In this case, the server generates a random 256-bit value which will be used as HMAC key (session authentication key) for data authentication with this client (mostly in step 2 of the project, but you have to do this in step 1). The server encrypts this HMAC key with the RSA public key of the client. Then, the server signs this encrypted HMAC key together with a positive acknowledgment message using server's own private RSA key. After these operations, the server sends the encrypted HMAC key, the positive acknowledgment message and the signature to the client. Both positive (including the encrypted HMAC) and negative acknowledgments are signed by the server. The server should keep track of the session authentication keys for each client after a successful authentication phase (to be used in Step 2).

When the client verifies the signature on the server message, he/she makes sure that the sender of this message is the valid server. In this case, the client will decrypt the HMAC key using his/her own private RSA key and store it in the memory to be used in Step 2.

Authentication protocol may fail due to various reasons and cases (such as wrong keys, modified messages, etc.). If authentication protocol fails, the connection must be closed, but of course, connection/authentication can be initiated again through the GUI. We will test various failed authentication cases during demos; so, please test your own codes accordingly.

So far, it should be clear that the client needs to load the server's RSA public key, his/her own public key and encrypted private key. Moreover, the server needs to load its own RSA public/private key pair from the file system, before the connections. To do so, the client/server may browse the file system to choose the key file(s). Unlike the client private key, the server will read and use its private key in clear.

Moreover, the server will need the client public keys as well; they are to be loaded from the corresponding client public key files provided with this project document. Here, the server will not browse each client public key file; instead, they have to be automatically loaded by matching the usernames and the filenames during the protocol. However, the main repository folder needs to be set by browsing the file system at the very beginning, before any client connections.

**Disconnection**

When a client disconnects from the server by pressing a disconnection button or by closing the GUI form window, he/she must clear his/her RSA private key value from the memory (not from the file system). After disconnection, the same user may want to login again by connecting and running a brand-new authentication phase.

## Provided RSA Keys and Other Secret Info

*Server* has:

- *server_pub_prv.txt*: This file includes Server’s <u>public/private key pair</u> in *XML* format.
- Server also has a trusted public key repository which has *c1_pub.txt, c2_pub.txt, c3_pub.txt ... c9_pub.txt*. These files include <u>public keys</u> of nine different clients. These public keys are also in *XML* format. This does <u>**not**</u> mean that your program will be tested with at most nine different clients. These files are provided to you just for your testing purposes. We can test your programs with any number of clients.

*Client* has:

- *server_pub.txt*: This file includes Server's public key in *XML* format.
- *username_pub.txt* and *enc_username_prv.txt*: These files include *username's* public key and encrypted public/private key in *XML* format, respectively. In the project assignment folder, we provide *enc_c1_pub_prv.txt, c1_pub.txt, enc_c2_pub_prv.txt, c2_pub.txt, enc_c3_pub_prv.txt, c3_pub.txt ... enc_c9_pub_prv.txt, c9_pub.txt.* This does **not** mean that a client can only have a username "*c + an integer*". A client can have any username, except it cannot contain an underscore ("_") character (input check this).
- The usernames of the clients are *c1*, *c2*, *c3*, …, *c9*.
- The passwords of the clients are *pass1*, *pass2*, *pass3*, …, *pass9*, respectively.
- In the demo, we can use different usernames and passwords.

## Project Step 2 (Due: May 20, 2021, Thursday, 21:00)

In the second step of the project, secure file storage and relaying mechanisms will be implemented on top of the first step. In this step, after secure login (authentication), the authenticated clients will be able to upload/request/download any type of file (text, executables, pdf, word, video, audio, etc.) of any length (the files can really be very big) to/from the server anytime. All the messages and files exchanged between client and server are encrypted and authenticated. Moreover, the server stores the files only in encrypted form. When requested, the necessary keys and the encrypted files are relayed to the requesting user.

A particular client can request a file owned by himself/herself or another client. The main access control decision is given by the owner of each file. Thus, the server is just a storage and acts as a broker when a file request is issued. The server does not keep the keys used for file encryptions. However, the server must keep the authenticated online clients' list up-to-date and must know the ownership information of each file to process the requests.

The folder that the server stores the uploaded files must be specified at the server GUI by browsing the file system. The server stores the files in an encrypted form so that it does not have access to the plaintext content of the file uploaded by a client. *Upload*, *Request* and *Download* operations are explained in the following subsections.

### Upload and Store a File Securely

Each client can upload any type of file (text, executables, pdf, word, video, audio, etc.) of any length (the files can really be very big) to the server anytime. In order to upload and store a file to the server in a confidential and authentic way, firstly the client chooses the file to be uploaded by browsing his/her file system. In order to secure the file transfer, the client generates two random numbers: (i) a random 256-bit value to be used as AES key and, (ii) another 128-bit random value to be used as IV for AES encryption in CBC mode. These random values must be generated using cryptographically secure random number generators, which are available in most crypto libraries.

The files to be uploaded will be encrypted in CBC mode by the client using the AES key and the IV mentioned in the previous paragraph so that the server will not be able to read the plaintext file content while it is being stored. The server will overwrite the name of the files that are uploaded using a naming convention (mentioned below) to guarantee their uniqueness. Then, the encrypted file will be authenticated using HMAC algorithm. Using the previously generated 256-bit random HMAC key (session authentication key generated in Step 1), the client generates the HMAC value of the encrypted file. The hash algorithm used in the HMAC is SHA-512.

After that, the encrypted file and the HMAC value of the encrypted file are concatenated and sent to the server. When the server receives them, the received HMAC value of the encrypted file is verified using the session authentication key generated in Step 1 (remember that SHA-512 algorithm is used as the hash algorithm of HMAC). If the verification is not successful, the client must be informed by the server with a signed message and the file is not to be stored in the server. If the verification is successful, it means that the encrypted file content was not changed and the encrypted file can be stored (note that HMAC value is not stored).

In the upload process on the server side, the server stores the file as ciphertext by using a naming convention to guarantee uniqueness. The naming convention consists of two parts: "client's username" and "an incremental file index" separated by underscore character (e.g., “c1_0” for the username: “c1” and the file index “0”).  The file indices start from 0 and they are incremented by 1 as new files are uploaded by the same client. File extensions will not be used for naming purposes in the server-side. The server should keep track of the owners of the uploaded files and this naming convention actually allows it besides guaranteeing uniqueness.

At the end of this procedure, the client must be informed by the server with a signed message that contains the filename that the server assigned. The client must verify the signature on this message, and it should keep track of the original filename, assigned filename and the corresponding AES key and IV that were used to encrypt the files uploaded. Store these in a local key file so that after disconnection, the client can restore them. Please use your imagination in order to secure this local key file.

**Request and Download an Encrypted File Securely**

Each authenticated client can request to download a file stored at the server side. This process has four stages. A simple overview is given below, but please refer to explanations given later for the details.

<u>Requesting Client:</u> X,     <u>File Owner Client:</u> Y,          <u>Server:</u> S          <u>File:</u> F

1) X → S: download request for F using its name that was assigned by server (signed using X's RSA private key)
2) S → Y: ask for permission to send F to the client X

3) Y → S: F's AES key || IV || F's original filename  sent (encrypted with RSA and authenticated with HMAC)
4) S → X: relay message 3 and the encrypted file (all signed using RSA)

In the first stage, the requesting client sends the requested file name to server. To do so, firstly the requesting client signs the filename using her/his private key. After that, the filename and the signature are sent to the server. After receiving this information, the server will verify the signature using client's public key. If everything is OK, then the protocol continues with the second stage. Otherwise, the server informs the client about the failure via a signed message and stops the protocol.

In the second stage, the server first checks whether there is a stored file with that filename and the owner of the file is connected to the server. If there is no such file or the owner is not connected, then the server sends a signed message to client explaining the fact and the protocol finishes.

If the file exists and the owner is connected, the server checks whether the file being requested actually belongs to the requesting client herself/himself. If this is the case, an automatic permission is granted to download the file and the protocol continues with the fourth stage without relaying the AES key and IV. The server signs the encrypted file that is requested. Then, it appends the signature to the encrypted file and sends to the client. After that, the client decrypts the file and stores locally.

If the file being requested does not belong to the requesting client:

- The server should ask the permission of the file owner (stage 2). The server sends four parameters to the owner for asking his/her permission to send the file to the requesting client: the filename that is requested, the username of the client who requests the file, the public key of the requesting client and the generated HMAC value of the concatenation of the first three parameters. The HMAC key to be used here is the session authentication key generated in Step 1 of the project and the hash algorithm used in the HMAC is SHA-512.
- When the client receives the previous message, the received HMAC value is verified (remember that SHA-512 algorithm is used as the hash algorithm of HMAC). If HMAC is not verified, the protocol should stop and a signed error message is sent to the server. If HMAC is verified, then the client shows and approves/disapproves file sharing request through GUI (depending on your imagination and creativity). If the client gives the permission, he/she encrypts the AES key and the IV (used to encrypt the uploaded file), and the original file name using the requesting client's public key; this way, the server does not get the AES key and cannot decrypt the file, and even cannot learn the original file name (yes, we do not trust server). Then, the HMAC value of the concatenation of (i) encrypted AES key, IV, original file name, and (ii) a positive permission message is generated. After that, (i) encrypted AES key, IV, original file name, (ii) the positive permission message, and (iii) the HMAC value are

concatenated and sent to the server. If the client does not give a permission, a negative permission message is sent together with its HMAC value.

- Then, the server verifies the HMAC of the received message. It then checks whether the permission message is positive or negative. If it is negative, the requesting client is notified with a signed negative permission message. If it is positive, the server concatenates the encrypted AES parameters, encrypted original file name and the encrypted file, and then sign them. After that, all these encrypted data and the signature are concatenated, and sent to the requesting client.
- In the last stage of the protocol, the requesting client receives the permission result and, if approved, the file, original file name and the key. The requesting client first verifies the signature and check whether the permission is granted. If it is granted, it decrypts the received AES parameters and original file name using his/her own 4096-bit RSA private key and use AES key and IV for decrypting the received encrypted file. After the file is decrypted, it should be stored in a local hard drive with the original file name. If the permission is not granted, just show it in the GUI.