CS432 Computer and Network Security
Spring 2017

Term Project

Secure File Transfer Application

Project Step 1 Due: March 29, 2017, Wednesday, 21:00 (to be submitted to SUCourse)
Demos: March 31, 2017, Friday (Schedule will be determined later)

Project Step 2 Due: May 10, 2017, Wednesday, 21:00 (to be submitted to SUCourse)
Demos: Time and Schedule will be determined later

Submission and Rules

- In both steps of the project, you can work in groups of at most 3 people.
- All group members should appear at the demos.
- Both of the group members should submit the complete project in each step to SUCourse. No email submissions please. Make sure that both group members submit the same version.
- The submitted code will be used during the demo. No modification on the submitted code will be allowed.
- You have to submit a project report together with the codes in both steps. In your project reports, you are expected to document and describe your implementation in detail (order of protocol messages, communication ports, structures of protocol messages, security related and cryptographic details, problems you encountered and solutions you came up with, etc.). If your program is lacking in some requested functionalities, clearly explain them. This document should also serve as a user's manual.
- Delete the content of debug/release folders in your project directory before submission.
- Create a folder named ***AuthServer*** and put your authentication server related codes here.
- Create a folder named ***FileServer*** and put your file server related codes here (only for second step of the project).
- 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. *mcandan_Candan_Omer*). Put your *Server* and *Client* folders along with your project report into this folder.
- Compress your *XXXXX_Lastname_Name* folder **using ZIP**. Please **do not use** other compression utilities like RAR, 7z, ICE, bz2, etc.
    - You can download winzip here: www.winzip.com Trial version is free!
- 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).

Introduction

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

The *Authentication Server* module:

- authenticates the clients,
- distributes file server access tickets when clients request

The *File Server* module:

- stores the files and distributes the requested files to clients in a secure and authenticated way.

The *Client* module:

- gets authenticated to the Authentication server,
- connects to the File Server to upload, download files in a secure and authenticated way using the tickets that is obtained from the Authentication server.

Roughly, the authentication of the clients to the Authentication server is 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 servers and client should be reported in text fields at their GUI. 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)
- Verification results (digital signatures, HMAC verifications)
- File transfer operations and file details
- HMAC values, session keys, etc. in hexadecimal format

Project Step 1 (<u>Due: March 29, 2017, Wednesday, 21:00</u>)

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

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 user names, 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 private key of each user is given in encrypted form. The encryption is done using AES-128 in CFB mode. For the key and IV of this encryption, SHA-256 hash of the password is used, the first half of the hash output, i.e. the byte array indices [0…15], being the key and the second half of the hash output, i.e. the byte array indices [16…31], 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. Firstly, the user enters IP address and port number of the server, as well as his/her username and password from the client GUI. The encrypted 2048-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 paragraph. 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>.

After the decryption of the private key, the client connects to the server and he/she sends an authentication request to it, together with the username (not the password!). This request initiates the challenge-response protocol. To authenticate the client, server sends a 128-bit random number to this client. Then, 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-256. 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 sends a positive acknowledgment. Both positive and negative acknowledgments are signed by the server. When the client verifies the signature on the acknowledgment, he/she makes sure that the sender of this acknowledgment message is the valid server. After a failed authentication, the client will try again.

We will test various failed authentication cases (such as wrong keys, modified messages, etc.) 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 selected by browsing the file system at the very beginning before any client connections.

Disconnection

When a client disconnects from the server (by pressing a disconnect button or by closing the 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 running a brand-new authentication phase.

Provided RSA Keys and Other Secret Info

Authentication Server has:

- *auth_server_pub_prv.txt*: This file includes Authentication Server's <u>public/private key pair</u> in *XML* format.
- *file_server_pub.txt*: This file includes File Server's <u>public key</u> in *XML* format (to be used in Step 2).
- Authentication 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:

- *auth_server_pub.txt*: This file includes Authentication Server's <u>public key</u> in *XML* format.
- *file_server_pub.txt*: This file includes File Server's <u>public key</u> in *XML* format (to be used in Step 2).

- *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.*
- The usernames of the clients are *c1, c2, c3, …, c9.*
- The passwords of the clients are *pass1, pass2, pass3, …, pass9*, respectively.

File Server has (only for step 2):

- *file_server_pub_prv.txt*: This file includes File Server's public/private key pair in *XML* format.
- *auth_server_pub.txt*: This file includes Authentication Server's public key in *XML* format.

Project Step 2 (Due: May 10, 2017, Wednesday, 21:00)

In the second step of the project, file storage mechanisms will be implemented on top of the first step. In this step, after secure login (authentication), the authenticated clients will request tickets to access the File Server. After receiving an access ticket, a client can either upload a file to the File Server or download a file from it. File can be of any type (text, executables, pdf, word, video, audio, etc.) of any length (the files can be large).

Acquiring an Access Ticket

After a client is authenticated (using the mechanism of the first step of the project), then it can request a ticket from the authentication server. The purpose of this ticket is to grant access to the file server. When a valid and authenticated client requests a ticket, the authentication server prepares it in the following way. As a first step, the server creates three cryptographically secure random numbers to be the encryption key, IV and HMAC key for the communication between the client and the file server. The session key and the IV are both 128-bit numbers while the HMAC key is a 256-bit number. These values and the client's username are concatenated to form a plaintext version of the ticket (T_{plain}). Then, the server performs three separate operations:

- signs T_{plain} with its own private key: $S_{PrivA}(T_{plain})$
- encrypts T_{plain} with client's public key: $E_{PubC}(T_{plain})$
- encrypts T_{plain} with the file server's public key: $E_{PubFS}(T_{plain})$

The results of these operations will be concatenated into the final form of the access ticket. This ticket will be sent back to the requested client.

Upload 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 big) to the file server. In order to upload a file to the file server in a secure and authentic way, firstly the authenticated client requests a ticket from the authentication server to access the file server. Upon receiving the ticket, the client decrypts the relevant part of the ticket and verifies it with the signature of the authentication server. Then the client sends the ticket to the file server to be validated as well. The file server decrypts the relevant part of the ticket, verifies it with the signature of the authentication server and also checks if the client's user name matches with the decrypted content. If the decryption operation fails or the content of the ticket is suspicious, then the ticket is rejected. If the file server rejects the ticket, the client cannot perform upload operation.

Upon receiving positive acknowledgement from the file server, the client chooses the file to be uploaded by browsing his/her file system. The file to be uploaded will be authenticated using HMAC algorithm. Using the HMAC key from the ticket, the client generates the HMAC value of the file. The hash algorithm used in the HMAC is SHA-256. After that, the file, together with its file name, is encrypted using AES algorithm in CBC mode with 128-bit key size. The key and the IV used here again come from the ticket. At the end, encrypted file and the HMAC value of the plain file are concatenated and sent to the server.

When the file server receives the file, it decrypts it with the session key and IV coming from the access ticket. Then the server checks the integrity of the file by applying HMAC SHA-256. If the verification fails, then the server sends a negative acknowledgement back to the client. If the verification holds, then the server stores the file in plain text and sends a positive acknowledgement back.

All positive/negative acknowledgments sent by the file server must be signed by it and verified by the client.

Download a File Securely

The download operation is somewhat symmetrical to upload operation. Each authenticated client can download files. In order to download a file, the client first requests an access ticket from the authentication server. Upon receiving the ticket, the client decrypts the relevant part of the ticket and verifies it with the signature of the authentication server. Then the client sends the ticket to the file server to be validated as well. The file server decrypts the relevant part of the ticket, verifies it with the signature of the authentication server and also checks if the client's user name matches with the decrypted content. If the decryption operation fails or the content of the ticket is suspicious the ticket is rejected. If the file server rejects the ticket, the client cannot perform download operation.

After that, the client encrypts the name of the file that is to be downloaded from the file server. For this encryption, the client uses AES algorithm in CBC mode with 128-bit key size. The key and the IV used here again come from the ticket. The encrypted file name is then sent to the file server.

The file server decrypts the message from the client to find the name of the file. Again, for this decryption AES algorithm in CBC mode with 128-bit key size and the key and the IV used here again coming from the ticket. If the file with the given file name does not exist on the file server then the server sends a negative acknowledgement.

If the file exists, firstly it will be authenticated using HMAC algorithm. Using the HMAC key from the ticket, the file server generates the HMAC value of the file. The hash algorithm used in the HMAC is SHA-256.

After that, the file, together with its file name, is encrypted using AES algorithm in CBC mode with 128-bit key size. The key and the IV used here again come from the ticket. At the end, encrypted file and the HMAC value of the plain file are concatenated and sent to the client.

When the file is received by the client, it will be decrypted with AES algorithm in CBC mode with 128-bit key size. Then, the integrity of the file will be verified by HMAC SHA-256 operation on the decrypted file. If the file verification holds, then the file will be stored in the clients file system. If not, the received file content should be discarded.

Programming Rules

- Preferred languages are C# and Java, but C# is **strongly** 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.
- Your program should compile and run.
- We will test your programs using at least two different computers. Please make such tests as well before submitting your project.

Good luck!

Albert Levi
Ömer Mert Candan
Halit Alptekin