Assignment #3: VPN

EECE 412

October 15, 2014

GROUP #8
Ellina Sergeyeva (Gulnaz)
Kelvin Au (XXX SoRA)
Roger Soderling (Sudo Nim)
Sarah Louise Leong (l5p8)

Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

I. Program description

The program is packed as a runnable JAR file and is stored in the Git repository. Before launching the program, make sure that Java is installed on a machine. To launch the program, double click on the vpn_group8 file. The steps to run the program are as follows. The code of this program is on GitHub https://github.com/kelvinau/VPN-Implementation.

1. Select the user type, either as a client or a server, and click "Select".
2. If server is chosen, provide a port number and a shared key for encryption and click "Connect".
3. If client is chosen, provide a port number, IP address of the server and a shared key for encryption. Port number and shared key must be similar to the one provided on the server side. Click "Connect".
4. Provide a message in the message textbox to send over to the other end and click "Send".
5. If a message is available from the other end, the message will appear in the received message textbox.
6. To single step through the communication and message exchange process between the client and the server, click "Continue". The steps will be displayed on the program log textbox.

II. TCP Connection and Data Protection

Communication between the client and the server is done using socket programming. The server program initiates a specific port on the machine that it is currently running on. The client program uses that same port to establish a TCP connection with the server. The client also needs an IP address (or host name) of the server. The client socket connects to the server socket, which is listening for connections on the specified port. When the connection is established, both client and server have their own input and output streams initialized to receive and send data. On the client side, the client program continuously listens for incoming data. The received data is then displayed in a dedicated field on the GUI. The server side acts in a similar fashion: the server listens for incoming messages, and displays any received messages in the GUI. Message transmitted between the client and the server is protected using the encryption method in part C. Once the message is encrypted, the cipher text is sent to the other end of the socket. The receiver decrypts the cipher text using the same encryption method but following the opposite order to obtain the message.

III. Mutual authentication and key establishment

We have implemented Perfect Forward Secrecy protocol with RSA asymmetric keys to achieve mutual authentication and session key establishment. We use three sets of asymmetric keys: data key is for encrypting data (RSA), signature key for signing (SHA1 with RSA) and final key for final encryption (RSA). The data and signature keys are 1024 bits long. The length of the final key is data key size + signature key size + 88 bits. In total, the size of the final key is 2136 bits.

To explain our approach, let us give an example. Let the client be Alice and the server be Bob. First of all, Alice sends her session ID (RA) to Bob. In response, Bob will send his session ID (RB) concatenated with Alice's session ID and his secret exponent b. Overall, this data will be encrypted with Alice's public data key, then signed with Bob's private signature key, and then encrypted with the Alice's public final key. In other words, Bob will send to Alice the following message:

$$R_B, \{[\{R_A, g^b mod\ p\}_{Alice's\ Public\ Key}]_{Bob's\ private\ key}\}_{Alice's\ Public\ Key}$$

When Alice receives this message, she will decrypt in the reverse order and send the following message in response:

$$\{[\{R_B, g^a mod\ p\}_{Bob's Public\ Key}]_{Alice's private\ key}\}_{Bob'sPublic\ Key}$$

where a is Alice's secret exponent. The RB, $g^a mod\ p$ data is first encrypted with Bob's public data key, then signed with Alice's private signature key, and, finally, encrypted with Bob's public final key.

Following the Diffie-Hellman key exchange, both Alice and Bob can compute the shared session key $g^{ab} mod\ p$. That ensures that every session has a fresh session key.The values of a and b are randomly generated with a secure random generator, so that Alice and Bob have fresh secret values for every session. The values of g and p are hardcoded, since we assume that they are publicly known. Session IDs are produced by taking 130 bits from secure random generator. These 130 bits are encoded in base 32, which means that every 5 bits form the random generator produce a string of 26 characters. In the end, session IDs are obtained by concatenating all such strings.

IV. Encryption

The message is encrypted by Advanced Encryption Standard (AES) with Cipher-block Chaining (CBC), using double hashing for the shared secret value, i.e. key, with Caesar ciphers on each character of the plaintext. The reason for choosing CBC as the mode of operation is because it uses a chaining mechanism. Decrypting a block of cipher text depends on all the preceding cipher text blocks. A single bit error can affect the decryption of the cipher text, which gives its strong confusion property to encrypt the VPN message. Double hashing is used to make the hash value strongly dependent on several parts of the key, which strengthen its diffusion property. The encryption steps are as follows.

1. The key can contain any characters because it takes the ASCII value of the characters. The key is hashed into a 32-bit MD5 string, and then hashed into an integer array with a size of 10. The function of hashing the MD5 value is hash(i) = ASCII(i) + ASCII(ii) + hash(i-1), where hash(0) = ASCII(i) + ASCII(ii). The space size of using double hashing is$(2^{31} - 1)^{10} = 2^{310}$, which is very large that it takes thousands of years to test the key.

2. The plain text is split into segments and encrypted separately. Each character is XORed using CBC, with the encryption of Caesar cipher. The function is c(i) = Encryption(character XOR c(i-1) , key), where c(-1) = initial vector IV, a random number generated with a range $[0, 2^{30}]$. The hashed integer array is used for Caesar cipher to shift the characters. Each element of the array shifts only one character. It goes from hash(0) to hash(9), then go back to hash(0).

3. The characters have an assigned value calculated from Step 2, which is then converted into binary number. 0's are attached to it to make the number with a length of 40. The cipher text is created when all the binary numbers are concatenated.

The decryption of the cipher text is the reverse of the encryption method.

V. Real-world VPN product

For the authentication aspect of the program, Transport Layer Security (TLS) would be used to ensure the security of the channel. This would prevent eavesdropping and tampering.

For the encryption algorithm, AES encryption with Counter mode (CTR) would be suggested, as one of the concerns would be the speed of the message transmission. CTR provides good parallelism, which allows for fast message transmission speed. However, CTR would need a unique IV for each message. IV would be generated uniquely and randomly, which would require a database to save the used IV in order to prevent using the same IV more than once.

For the encryption key size, a 128-bit key would be suggested for AES encryption. 128-bit key provides an almost equivalent degree of security as 192-bit key and 256-bit key but at a lower cost. For an adversary to perform a brute force attack, they would need to try 2^{127}ways on average to obtain the right key which already requires a lot of time and computer resources. Hence, there is no need to use larger key size for AES encryption.

For key exchange between two or more parties, Elliptic Curve Diffie Hellman would be suggested as it provides a secure way to exchange keys or to generate keys that are able to encrypt subsequent messages with AES-CTR encryption. This method allows two parties to exchange keys without explicitly sending the key itself on the network, Hence this reduces the possibility of a third party stealing the encryption key.

VI. Program description

Our VPN software is written in Java. The program contains 1010 lines of code and the size of the executable file is 33KB. The program is divided to four major architectural components which are as follows.

A. Graphical User Interface (GUI)

This module implements the graphical user interface of our program and glues together other modules. The GUI module is the starting point of our program. When the executable is launched, the GUI.java creates a panel that contains text fields and buttons. When a user clicks on a button, a corresponding action is performed and a relevant package is called. For example, when the "Connect" button is clicked, the TCP Connection module is invoked to initiate a TCP connection. When the "Send" button is clicked, the Encryption and TCP Connection modules are invoked to send encrypted message to the receiver. The Authentication module is invoked during a connection set up. The GUI module also keeps a log of messages being exchanged between the client and server. It

allows users to single step through the process of message encryption and decryption. The GUI module is represented by the gui package and contains the Gui.java file.

Input: User actions by clicking the button.
Output: The GUI panel, calls to other packages.

B. TCP Connection

This module establishes a communication channel between a server and a client to authenticate users or pass messages to each other by socket programming. To do so, the server provides a port number while the client provides the same port number and IP address of the server. The client socket then connects to the server socket. The client and server continuously listen for incoming data on the specified port. If a message is available on the data stream, TCP connection module interacts with the Encryption module to either encrypt or decrypt messages. If a cipher text is received by either of the endpoints, it is decrypted and displayed on the GUI. This module corresponds to the clientServer package and the VPN.java file in our program.

Our original plan was to have the TCP connection module run on two modes which are the mutual authentication mode and message encryption mode. If a connection is established between a client and a server, the TCP connection is run on a mutual authentication mode. This triggers the mutual authentication module to authenticate both client and server. Once the mutual authentication module is done, it signals to the TCP connection module to run on message encryption mode to handle the encryption and decryption of messages sent between both users. However, we were unable to integrate this functionality with the rest of program due to the fact that the authentication data was corrupted when transferred through a TCP connection. We established a reliable way to send and receive strings; however, when sending char arrays or byte arrays and converting them to strings, some data was corrupted, so that the session key could not be generated.

Input: Either (a) or (b)
(a) On the client side: IP address, port number and message
(b) On the server side: port number and message
Output: A TCP sockets connection between the server and client. Messages received by client and server.

C. Data encryption

This module handles the encryption and decryption of the message transmitted on a TCP connection. When a user sends a message via a connection port, the message is encrypted before it is sent out. When a user receives a message via a connection port, the message is decrypted and displayed on the appropriate textbox on the GUI. This module corresponds to the encryption package that contains the encryption.java file.

Input: A shared key, a cipher text or plain text
Output: Encrypted message or decrypted message

D. Mutual authentication

This module checks the authentication of server and client during an established communication between both parties. It is triggered after a client attempts to connect with a server. Once a TCP connection is made and run on a mutual authentication mode, the module initializes a key sharing session between the client and the server by passing session keys and a string that corresponds with their private key. For each new TCP connection, a fresh session key is generated. If the mutual authentication succeeds, it signals to TCP connection to run on message encryption mode to initialize a message sharing session. This module corresponds to the authentication package that contains the authentication.java file.

Input: None. The module is autonomous with respect to input data. It needs a connection though to exchange messages in order to mutually authenticate users and establish a session key.
Output: Shared session key