COMS 4180 Network Security Programming Assignment 2

125 points
Due Monday February 29, 2016, 2016 10:00pm Eastern time.

- This assignment is to be done individually.
- NO LATE SUBMISSIONS WILL BE ACCEPTED.
- The code for the programming problem must compile and run from the command line in linux on the computer science department clic machines - DO NOT assume the use of a specific IDE such as Eclipse to run the code.
- Your code must be commented. Uncommented and poorly commented code will lose points.
-

Submission Instructions

- Assignments will be submitted via Canvas.
- Submit a zip file containing a tar file of your source code, makefile and readme file, and the certificates and public-private keys you used when testing. Do not include any executables in your submission.
- The zip file name must be of the form UNI_#.<extension>, where "UNI" is your UNI and "#" is the number of the assignment, and the extension is zip or tgz.
- Please put your name at the top of each file submitted, including as a comment at the top of each file containing code.
- The Makefile is mandatory for C, C++ and optional for JAVA. If you use JAVA and no makefile, your code will be compiled by typing ”javac *.java” If you include a Makefile, your code will be compiled by typing ”make”
- The README file contains (1) any instructions for your code, (2) **the step by step instructions** you followed for creating the certificates and **proof that you tested 2-way authentication** by showing the messages containing the certificate from the server to the client and from the client to the server. As proof either (I) cut and paste of output from tcpdump using the most verbose output and/or some combination of hex and ASCII view of the output or (II) take screen shots of the messages in wireshark as proof. You do no t need to include the entire expanded output from wireshark - the summary line showing the info column and an expansion of the packet (middle portion of the screen) to the extent that the top few lines under "+ certificate" are visible is enough.

In this problem a client and server will establish a TLS connection that uses **2-way authentication**. The client will be able to send and retrieve files to/from the server. The client will send a hash with each file and verify the hash of a retrieved file. The client will optionally encrypt/decrypt the file with AES in CBC mode. The problem is intended to familiarize you with the TLS libraries and tools for creating certificates. Learning how to use the libraries and set up certificates is part of the assignment. You are required to submit step-by-step instructions for how to create certificates in your README file.

You may use any programming language provided the code will run on clic without requiring the TAs install additional libraries. If programming in C/C++, use the openssl library. Openssl includes command line functions for generating certificates. If programming in JAVA, use the API for tls/ssl. For JAVA, look at the keytool utility for generating certificates. The default settings in both C/C++ and JAVA are for one way authentication. **Be sure to establish the connection to use two-way authentication.** You must generate the appropriate certificates so that clients and servers can trust each other and create the appropriate certificate stores. For mutual authentication to take place, the certificates must be imported into the appropriate stores. For this exercise, **certificates may be self signed** or you may create a CA and sign them. For this homework, there is no need to get a certificate signed by a real CA. The certificates may use (RSA and SHA256) or (ECDSA and SHA256) as the signature and hash

algorithm in the certificates. If using ECDSA, you may specify any EC curve that is available when creating the certificate if the tool you are using requires the name of a curve be specified.

You may use any online example/tutorial you find for creating certificates as long as you clearly document the steps you followed in your readme file. For creating the TLS sockets, you may find online code samples for setting up TLS sockets and use them as a starting point for your code, just remember to clearly comment the code and include error handling.

Server:
When the server is started it will take as command line arguments

- the port number on which to listen
- any other arguments needed in order to use the certificates required for TLS

Client:
When the client is started it will take as command line arguments

- the server's name or IP address; The client and server will be executed from different machines so do not use ”localhost” in place of IP address or hostname. If an IP address is used, assume it is IPv4.
- the server's port number
- any other arguments needed in order to use the certificates required for TLS

Once the client has established a connection to the server, it will prompt the user to enter a command on a single line consisting of the following in the order listed here:

- the string "put" or "get" or "stop" to indicate if the client will be sending a file (put) to the server or retrieving a file (get) from the server or exiting.
- In the case of "get" or "put", the following is also required:
  (1) the filename to get or put
  - For "put": if a path is included in the filename, it may either be the full path or relative to the directory from which the client executable is run; if there is no path in the filename, the client will look in the directory from which it was executed for the file.
  - For "get": if a path is included in the filename, it may either be the full path or relative to the directory from which the server executable is run; if there is no path in the filename, the server will look in the directory from which it was executed for the file.

  (2) A one character flag of "E" or "N"
  - "E" means the client will encrypt the file with AES in CBC mode prior to sending it (in the case of a "put") or attempt to decrypt the file with AES in CBC mode upon retrieving it (in the case of a "get").
  - If "E", an eight character password must also be provided as an argument after the "E"
    You may assume any ASCII characters that can be entered via the keyboard are valid (the password is treated as 8 bytes). If you need to restrict what can be in a password, include the restrictions in your README file and print a message indicating the restriction when the client prompts the user for the inputs.

File processing:
"**put":**

1. The client will generate the SHA256 hash of the plaintext file.
2. If the file is to be encrypted, the client will use the password as a seed to a random number generator (RNG) to create a 16-byte AES key. Make sure you use a deterministic RNG library function so the same key is generated when given the same password. The client will then encrypt the file with AES in CBC mode. The client will create the IV used in CBC mode. You may use a constant or create an IV each time a file is encrypted.

3. The client sends the file (with the IV prepended to it if the file is encrypted) and the hash to the server.
4. The server will write the file (still with the IV preprended if the file is encrypted) and its corresponding hash to the directory from which the server was executed. These are saved as two separate files (the file and the hash). The hash will be saved in a file with the same name as the file being transmitted with a ".sha256" extension added to the name. For example, if the file is abc.txt the hash file is named abc.txt.sha256

"get":

1. The client will send a request to the server asking for the file
2. The server will respond by sending the file and its corresponding hash. The server will always look for the hash with the same name and .sha256 extension in the directory as the file.
3. The client will decrypt the file if "E" was specified.
4. The client will compute the sha256 hash of the plaintext file
5. The client will compare the hash it computed to the hash that was received.
6. If the hash matches, the client will write the file (not the hash) to the directory from which the client was executed. In the case of a text file, the user will be able to read the file in another window or after the client stops to manually verify it was retrieved and decrypted. (The programs will be tested with ASCII and binary files when being graded.)
   If the hash did not match, the client will display a message to the user before displaying the prompt again and will not write the file to disk.

After the action ("get" or "put") is completed, a message is displayed indicating the action was completed or the action could not be completed (as described below). The client prompts the user again so the user can continue to send and retrieve files in any order until entering "stop".

- In the case of "get", if the file (or its hash file) the client requested does not have the appropriate read permission to allow the server to access it or the file (or its hash file) cannot be found, the server should return a message indicating the file cannot be retrieved without indicating to the client whether the file exists or not. For security reasons, if a file exists but is not readable by the client, the server may not want to let the client know the file exists and instead just return a generic message that the file cannot be retrieved. The client should be able to retrieve any file it writes to the server.
- In the case of "put", if the client cannot access the file or the file does not exist, the client should display a message indicating the file cannot be sent without indicating if the file exists or not.
- In either case, if any other error occurs (such invalid input parameters), an error message describing the problem will be displayed to the user then the prompt presented again.
- If the client attempts to decrypt a file that was not encrypted (detected because the call to AES-CBC will return an error message), no file will be written and the client will display a message indicating decryption failed then display the prompt again.
- If the client decrypts the file with a password that was not the same password used to encrypt the file, the computed hash will not match the hash retrieved from the server. The client will display a message to the user indicating the hashes did not match then display the prompt again.

The program must support using the get and put commands in any order, any number of times.

Example session:
In this example, ">" is the command line prompt.

```
server:
$ ./server 9955 <other command line arguments>

client:
```

```
$ ./client <server's IP or hostname> 9955 <other command line
arguments>
> put x.dat N
transfer of x.dat complete
> get x.dat N
retrieval of x.dat complete
>put y.dat N
Error: y.dat cannot be transferred
>get z.dat N
Error: z.dat was not retrieved.
> put x.dat E password
transfer of x.dat complete
> get x.dat E wrongpwd
Error: Computed hash of x.dat does not match retrieved hash
>get x.dat E password
retrieval of x.dat complete
> put abc.txt N
transfer of abc.txt complete
> get abc.txt E password
Error: decryption of abc.txt failed, was file encrypted?
>get abc.txt X
Invalid parameter "X"
> get N
Error: Missing parameters, a minimum of a filename and "N" or "E" is
required
>get x.dat E
Error: Missing parameters, "E" requires a password
> x.dat E password
Error: Invalid commands, options are "get" "put" "stop"
> stop

server:
$ (send signal to kill server if needed)
```

Error Handling:
When starting the client and server, the programs will be tested for handling of invalid/garbage/missing input. The programs must check the validity of the input parameters when starting (i.e. the IP address/server name, port number, any parameters needed for TLS) and exit nicely if anything is invalid, printing a message specifying the required input parameters before exiting. This includes but is not limited to missing parameters, improper values (length, type, value), out of order parameters. Any runtime error must also be handled by printing an appropriate message and exiting nicely. (for example, segmentation faults in C/C++ will result in a grade of 0). NOTE: Leaving one side of a socket open is NOT exiting nicely. For example, if the server side if a socket dies, the client's side should not print the default exception to the screen (such as occurs in JAVA when exceptions are not handled) or just hang.

Once running, the client must display informative error messages to the user before continuing if any of the commands the user entered are invalid or missing parameters. A subset of possible invalid inputs are shown above in the example session.