COMS 4180 Network Security Spring 2018 Programming Assignment 1

Due Wednesday Feb 14, 2018 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 Ubuntu 16.x or 17.x on the Google cloud account. Your code must be executable from the command line. DO NOT require that your code has to be run using a specific IDE, such as Eclipse.
- Google accounts and setup instructions will be given out within the next 7 days. You can start writing your code before getting your account.
- **Your code must be commented. Uncommented and poorly commented code will lose points.**

Submission Instructions (refer to the list of files at end of this document for more details)

- Assignments will be submitted via Courseworks.
- Submit a zip file containing a tar file of your source code, a Makefile, a readme file, and files containing the RSA keys you used for 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.

Programming Problem: (150 points total)
This problem involves using crypto libraries/functions for encrypting, hashing and signing data. Use existing library routines as opposed to writing your own functions for any crypto algorithms needed. You may use any programming language, C/C++ or JAVA may be the easiest. If using C/C++, the openssl library contains the crypto functions. If using JAVA, the JAVA crypto library has the required functions. There is no need to use a 3rd party JAVA library for the crypto. If using Python, consider cryptography or (older) pycrypto.
You may install any programming languages, their associated dependencies and useful shell commands on your VMs.

Overview
There will be a client and server with regular sockets (not TLS) between them.
The client and server have RSA keys (using 2048 bit modulus) that will be used in this process. You are responsible for determining how to generate RSA keys and store the RSA keys so the client and server can read the necessary keys. The client and server must not see each other's private key. Note: if running both the client and server on a single VM, it is ok for this assignment if the private keys are stored in separate files in the same directory. It is not ok to store an entity's public key and private key in a single file that the other entity reads to get to the public key.
Do no hard code the server's IP address and the port numbers the client and server use for the socket. The port numbers should be specified on the command line when starting the client and server. The server's IPv4 address should be specified on the command line with starting the client.

The client is capable of reading a file from disk and, based on command line input, do one of the following actions:

a. Use a password (entered on the command line) to generate a 128-bit key, K, encrypt the file with AES in CBC mode using K, encrypt K with the server's public key, send the encrypted K and the encrypted file to the server.

b. Sign the file using SHA-256 and its RSA private key then send the file and signature to the server. In this case, the client is not encrypting the file.
c. Sign the file using SHA-256 and its RSA private key, then alter the first byte in the file, then send the altered file and signature to the server. In this case, the client is not encrypting the file. The first byte in the file is altered by setting it to 0x00 if the byte was not already 0x00 and by setting it to 0xff if it was 0x00. The client will also print to the terminal in hex the original value and the altered value of the first byte. Note: This case will result in the server detecting the signature does not correspond to the file it received.

The server will be given a command line parameter indicating it is going to be decrypting a file or verifying a signature. When verifying a signature, the server will not be told if the client used case b or c. You are responsible for determining the proper RSA key components the server uses for decrypting the key and for verifying the signature.

- In the case where the server receives an encrypted file, the server will decrypt K then use K to decrypt the file and write the file to disk.
- In the case where the server receives a signature and file, the server will verify the signature on the file and print to the screen “signature is valid” if the signature is correct or “signature is invalid” if the signature is invalid.

The file may be any format so make no assumptions about the format. You should treat the file as bytes (not as some object) and must be able to support binary files. If necessary, you may assume the programs will not be tested with any file over 1 MB.

In the case where the server receives an encrypted file, the decrypted file the server produces will be compared to the original file when grading the program so it is recommended you diff the result with the original file when testing your programs. There should be no differences, including no differences for end-of-file.

Input Specifications:
Client:
The client accepts as input parameters the items listed below. The client will perform the action indicated by the single character (a,b or c). The client disconnects from the server after sending the data to the server.

Inputs:
The following information should be input in the command line when executing the client.

- Password: The 16 character password may contain any alphanumeric character (i.e. lowercase, uppercase and digits). Note: special characters are not included.
- filename: Clearly indicate in your README file if the path of the file provided as input must be the full path or relevant to the directory containing the executable. You may require that the file be in the same directory as the executable.
- Action: single character of a,b or c corresponding to the actions described in the Overview.
- server IPv4 address
- port number to use when contacting the server
- Necessary RSA key components: any inputs (filenames) to provide the client the necessary information for the RSA keys . Key components should be read from files and not have to be typed by the user.

Server:
The server takes the following inputs. The server will be started and wait for the client to connect. The connection should be torn down nicely and print a message to the terminal that the client has disconnected (no core dumps and no exceptions printed to the terminal).

Inputs:
The following information should be input in the command line when executing the server.
- The port number on which the server will listen for a connection from the client.
- mode: A single lowercase character of d or v. d means it will be decrypting a file, v means it will be verifying a signature.
- Necessary RSA key components: any inputs (filenames) to provide the server the necessary information for the RSA keys . Key components should be read from files and not have to be typed by the user.

Notes on padding when using a crypto library:
The standard for padding will result in 1 extra block being added when encrypting in CBC mode if the plaintext is an integer multiple of 16 (this extra block is padding that is needed so the recipient decrypting the data does not interpret the last block of the actual data as padding). This is done automatically by the library functions. If the plaintext was not an integer multiple of 16, the last partial block is padded and a full block is not added.

Error Handling:
Programs will be tested for handling of invalid/garbage/missing input. The programs must check the validity of the input parameters 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, with the exception that invalid POSITIVE NUMERIC values for RSA parameters may not be detectable until subroutines are called that uses these, at which point any detected error/exception should be handled by printing an informative message indicating the error and exiting nicely. 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.
What to submit for the program:
- Do not submit any executables
- client source code titled client.<ext> where <ext> depends on the language you used
- server source code titled server.<ext>
- The RSA files (RSA keys) with which you tested your programs
- any helper functions that are in separate files, including any program you wrote to generate RSA keys if you did not use an existing tool/openssl command line
- A makefile - mandatory for C, C++, 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"
- README file: (1) What you had to install on the VM, i.e. apt-get install <NAME> for everything you installed. (2) the steps you used to generate the RSA keys (3) how to run your programs. If you have files other than client and server, describe any helper functions that are in separate files - include a list of such files with one or two lines stating the purpose of each.

If you are using JAVA, you will need to use client and server as the class names (these are in lower case). If you are using python, use client.py and server.py as the names for the source code. If you are using C/C++, the executables produced by the makefile should be named client and server.