COMS 4180 Network Security Spring 2016 Programming Assignment 1

Due Monday Feb 15, 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.
- **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 Canvas.
- Submit a zip file containing a tar file of your source code, makefile and 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: (125 points)
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. Do not download and install 3rd party crypto libraries. Your code must run on the clic machine without requiring that the TA install additional libraries. You may use any programming language on the clic machines. C/C++ or JAVA may be the easiest. If using C/C++, the openssl library contains the crypto functions. (Note: the openssl version on clic does support SHA-256 even if the help message does not display it in the list of algorithms.) 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.

Overview:
There will be a client and a server with regular sockets (no TLS) between the client and server. The client will encrypt and sign a file, then send an encrypted key, the encrypted file and the signature to the server. The client and server have RSA keys (using 2048 bit modulus) that will be used in this process. The server decrypts the file, checks the signature and indicates whether the signature verification failed or passed. We will pretend that the server has malware on it that may sometimes replaces the encrypted file with another file. A command line argument given to the server will indicate if the file is replaced.
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: since you will be running both the client and server from your clic account, it is ok 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 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.

Specifications:
• Client:
The client accepts a 16 character password and a filename (full path if needed) as input parameters. the client will encrypt the file with AES in CBC mode. It also signs the plaintext of the file by hashing the plaintext file with SHA-256 then encrypting the hash with RSA using its private key. The client sends the

encrypted data and signature to the server. The file may be any format so make no assumptions about the format. If necessary, you may assume the programs will not be tested with any file over 1 MB. The client also encrypts the password used as the AES key with the server's public RSA key and sends it to the server. The client disconnects from the server after sending the password, file and signature.

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 in order to simplify the input.
- 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 just require that the file be in the same directory as the executable.
- server IP address or name
- 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 will be run in one of two modes, trusted or untrusted. In trusted mode, the server receives the encrypted password, encrypted file and signature from the server, decrypts the file and verifies the signature, writing the result (”Verification Passed” or ”Verification Failed”) to standard out (the terminal window). The server writes the unencrypted file received from the server to disk in the same directory from which the server application was executed. It names the file ”decryptedfile” (no file extension in the name).
In untrusted mode, we pretend that malware has replaced the received file with one called "fakefile" (no file extension in the name). The server performs the same steps as in trusted mode, except it uses fakefile instead of the file it received from the client.
You may have the server always write the file received from the client to disk and have the server read the received file or fakefile, as appropriate based on the mode.

The file decryptedfile 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.
You are responsible for determining the proper RSA key components the server uses for decrypting the password and for verifying the signature.

The TAs will be using files of size <= 1MB for "fakefile".

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 t or u. t means trusted mode, u means untrusted mode (file gets replaced).
- 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 the details of padding when using library functions:
1. When using AES from a crypto library, the function return value may indicate decryption failed and not produce a plaintext file. This occurs when the file it is attempting to decrypt has a length that is not an

integer multiple of 16-bytes and/or does not have the proper padding CBC mode uses (the encryption function takes care of adding the padding). So if you test with an arbitrary "fakefile" in untrusted mode, the server will be able to write "Verification Failed" at this point. However, when testing, you should encrypt some file (different from what the client will encrypt) with AES in CBC mode using the same AES key used by the client and use the ciphertext as "fakefile" to verify the client can correctly tell it was not the correct file based on the signature.
2. 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) the steps you used to generate the RSA keys (2) 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.