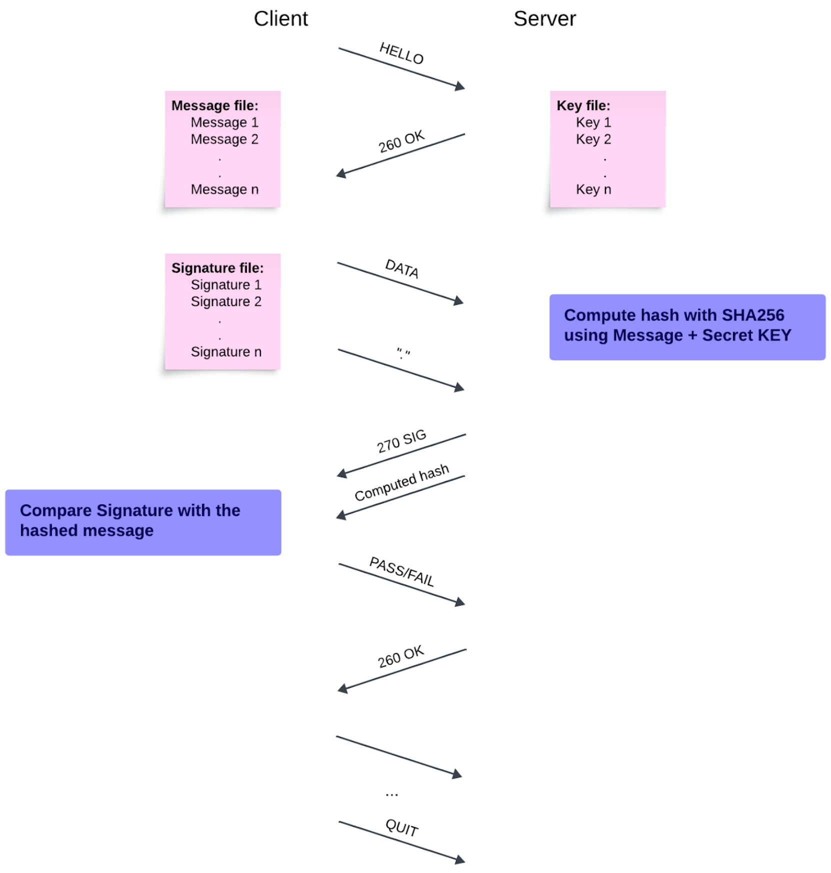
CS 352 Assignment 2: Message Validating Client and Server

In this assignment you will write a client and server that validates the integrity of text messages using a protocol based on secret keys. In this scenario, a client has downloaded a file from a 3rd party, untrusted source that contains several text messages from the server, for example, emails. The client also has a file of associated “signatures”, or hashes, of the messages, also from the untrusted source. The signatures prove that the server originated the messages, however, the client must verify these signatures.
In this project, the client will validate that the messages came from the server by assuming a secure channel, and then contact the server and have it provide the signatures. If the signatures match, the server must have originated the message. Note that in a more realistic scenario the server would have a pre-computed database of messages and hashes, however, for this assignment we’ll assume there is a trusted connection between the client and server so the server can compute the hashes on the fly.

Figure 1 shows an overview and example protocol exchange between the client and server.

Overview:

Figure 1: Example Protocol Exchange



One way hash functions:
A one-way hash function takes a long stream of input, say a text file, and produces a fixed sized hash value. For example, in SHA 256 it is a 256 bit number, and in MD5 it is a 128-bit number. The server takes a string of text (potentially long) and a secret key, and hashes them together. The resulting one-way hash validates that only the owner of the secret key produced the hash value. If the owner produced such a hash value, it “signed” the message.

You can find examples of using SHA 256 to generate hash codes at the links below:
https://docs.python.org/3/library/hashlib.html

https://www.geeksforgeeks.org/sha-in-python/

The protocol:

The client and server communicate via TCP with a port defined as a command line argument.
The protocol works by sending ASCII characters — not UTF-8 or unicode strings.
The client connects to the server and issues a “HELLO” on a single line.
The server responds with a “260 OK” string if it receives the “HELLO” message.
Then the client sends a Command. There are only two commands: “DATA” or “QUIT”.
Each time that the client wants to send a message to the server it will send the DATA command first. That means the client sends the DATA command separately and Msg 1 after that. Then for sending the next message it will send another DATA command.
The server checks if it received the DATA command, then it will compute the hash using the key and the message.
The server then sends the computed hash value and a “270 SIG” string to the client.
The client will compare the computed hash value with the signature value which is provided, and it will send the PASS or FAIL to the server.
Then the server sends the “260 OK” string once again when it receives the PASS and FAIL result from the client.
Server checks if it receives the QUIT command, so it will close the connection

A DATA Message:
The client sends the DATA command. Then it sends the ASCII text of the message.
The end-of-message is a single “.” (dot) on a line by itself.
At the end-of-message line, the server responds to the client with the response “270 SIG” on one line, then the SHA256 value as a hexadecimal number is ASCII on the next line.

A QUIT Message:
After the client sends a quit message, it closes the TCP socket and exits the protocol.

Escape codes:
When using in-band signaling, that is, sending control information on the same stream as data, escape codes are needed. Recall a ‘.’ on a line by itself signals the end of the message, it is not

part of the message data. We need a way to distinguish between a line with a single dot as part of the message, and the control signal that the data has ended.
To differentiate a dot on a line in the data of the message from the end-of-message code, a single dot in the message is “escaped” with a “\”. So a text message with a single dot would be sent as: “ \.\r\n ”. Further backslashes are escaped with a “\” as well. So a message with a “\.” in the data would be sent as “\\.”. Thus, when your code reads a line, it should unescape any strings with a ‘.’ and a set of \\’s.
Three message files have been provided to you. The messages in the advanced message file have multiple lines of message and dots in between. In those messages you have to differentiate between a dot in between and at the end of the message. Therefore, you will escape the dots in order to keep them as a regular character with no function (end of the message). That means you will escape the dots in the middle of the message and specify the end of the message with “ \.\r\n ”.

Message file format:
A message file is a sequence of message-lengths and messages in ASCII format.
A message length is an unsigned integer string followed by a control line feed (end of line in Unix).
A message is a sequence of ascii characters of the length of the message.

Signature file format:
The signatures, one for each message, are stored as hexadecimal numbers in a string format, one per line.

Key file format:
The secret keys, one for each message, are stored as strings, one per line.

Pseudo code for the client:

Start the program with the following arguments in order:

<server-name> <server-port> <message-filename> <signature-filename>

For example: python3 client.py localhost 7894 message1.txt sig1.txt

```
Open the message file from the name in the command line
While there is still more data in the message file:
    Read in one line
    Convert the string into the number of bytes
    Read in the number of bytes from the message file into a byte string or byte array
    Append the bytes of the message into an array of messages
Open the signature file from the name in the command line
While there is still more data in the signature file:
        Read in one line
Append the string of the signature into an array of signatures
Open a TCP socket to the server using the name and port from the command line
Send a “HELLO” message to the server
```

```
Read the response
If the response is not “260 OK”,  print an error and end the program.
Set a message counter variable to zero
Foreach message in the array of messages:
        Send the DATA command in one line on the TCP socket
        Send the message on the TCP socket
        Read a line from the server from the TCP socket
        If the response is not “270 SIG”,  print an error and end the program.
        Read another line from the server
        Compare the string from the server with the signature string stored in the array
                signatures for this message at the message counter number
        If the strings match:
            send a PASS message to server
        Else:
            send a FAIL message to the server
             Read a line from the server
        If the response is not “260 OK”, print an error and end the program.
        Increment the message counter
Send a QUIT message to the server
Close the TCP socket.
```

Pseudo code for the server:

```
Start the server with the following arguments in order:
        <listen-port> <key-file>
        For example: python3 server.py 7894 key.txt
Read in all the keys from the key-file.
Open a TCP socket on the port
When the connection competes, read a line from the returned connected socket
If the line is not “HELLO”, print an error, close the socket, and exit the program
While there are still more messages to validate:
Read the next line from the socket
Case statement on the string in the line # use case statements, one case for each command
   DATA:
                Start a new SHA 265 hash
            While there are more lines in the message:
                    Read a line from the TCP socket
                    Unescape the line
                    If the line is “.”, or it’s escaped equivalent, break from the loop
                    Add the line to the SHA 256 hash
                        Add the key to the hash, and finish the hash
                    Send the 270 SIG status code back to the client on one line.
                    Send a hexadecimal string value on the TCP socket of the signature on one line.
                    Read a line from the socket
                If the line is not either “PASS” or “FAIL”, print an error, close the socket and exit the program
            Send a “260 OK” string on the socket
        QUIT:
            Close the socket and end the program
        Default:
            Print and error, close the socket, and end the program
```

Grading:

Your program will be auto-graded. Please note that your code should work for all the three message files that have been provided.

Program names:

You must upload a zip file called `submission.zip` with 2 files, one called `client.py` and `server.py`. Your code must run from a main function, not the start of the script. E.g., something like:

```
def main():
    print(“Your code goes here")

if __name__ == "__main__":
    main()
```

Test 1:

We will run your server against our client. The server must follow the protocol and produce the correct hashes for all the messages.

Test 2:

We will run your client against our server. The client must follow the protocol and produce the PASS/FAIL responses for each of the messages.

Client and Server Outputs for Grading:

The client and server should print out every message they receive using normal print statements (to stdout). Each message should be on one line. For example:

- Every time you receive a message print it out on its own line
- So for example if you receive a “270 SIG” code from the server followed by its computed signature these should both be printed, each on their own lines on the client script.