Project 1

Vidal Zazueta, Nick Janes, Mingyang Liu

1. Project Description

A secure socket layer is a security protocol that provides privacy, authentication, and data integrity for communication over the Internet. In this project, we created a simplified version of a secure socket layer (SSL) that runs on 2 command prompts on a computer. The SSL we implemented contains various key components, such as key generation using the RSA ciphering algorithm, a Hash function, one-time key encryption, and the actual secure socket layer with communication between two socket processes.

The RSA algorithm that we implemented generates public and private key pairs of an arbitrary size and can also cipher and decipher short messages. In our approach, we grabbed the input value from prime\_size from the terminal input to help generate the key size. We created the keys using constructors and a method to generate them. In this method, we broke down our algorithm into more methods such as getting the relative prime of phi given an input. We also used several built-in Java methods such as getting the mod inverse of an input from the Big Integer class. To add on, we implemented our methods to encrypt, decrypt, verify, and sign a text using the keys we generated.

The Hash function we implemented employs various methods to operate. First, we grab the values, such as ndatabytes, ncheckbytes, etc, that are inputted and assign them to appropriate variables to be used in methods. We then created a method to mimic the formula given to us by the professor to use the values that were inputted. To add on, one of our methods generates a checksum to be added to the packet. We also created a method to create a one-time key of the same length as the packet and a method to encode the packet using the XOR operations on each bit with the one-time key. The method to decrypt the packet also uses the same one time key and the XOR operation to decode the packet. We also created a method to print out the packet into binary numbers as printing them out as a string would output symbols.

The One Time key that we implemented in the OneTimeKey java file is very simple. We take the 2 arguments inputted on the terminal, with the 1st being the key and the 2nd input being the text. We then convert these inputs into integers, and use the encryption and decryption methods we created on these integers.

Finally, we have the simplified version of a secure socket layer that we implemented. In this, we have both a client and a host. The client communicates with the server to initiate a handshake that communicates the client username, company, and the proposed one-time key from the client. In our project, the client gets these 3 bits of information from the mickey text file where we parse the information into assigned values to be used later for the handshake. In the handshake, we encrypt the username, sign the company with the client’s private key, and encrypt the one-time key with the server’s public key. It is then put into an encrypted handshake and sent to the server, where the server will extract the 3 bits of information.

Moving forward, the server will extract the 3 pieces of information from the handshake. It will decrypt the username with the server’s private key, then verify the company, and then decrypt it with the client’s public key, and close the connection if there is a company mismatch. With everything setup, the client and host can initiate communication and send messages. The client gets the values from the mickey text file while the server gets the values from the users text files. The client generates a packet that is then encoded by the proposed one-time key. It is then sent to the server, where the server decodes the packet with the same one-time key. The server then checks the checksum, and if they don’t match, the server closes the connection. If they do match, the server will modify the text; all uppercase characters will become lowercase characters and vice versa. The server then encodes the message with the same one-time key as before and sends the packet back to the client. The client receives the packet and decodes it with the same one-time key and displays the message. It will then state the amount of bytes that were written.

1. How to run the program

To run the program, you must first open a command prompt and then navigate to the location where the folder Project 1 is stored. Upon entering the folder, you can run all of the programs after you compile them. To compile the files, input the command; javac <fileName>.java for each file. After compiling the files you can run them as follows:

Starting with part A of the assignment, we have the RSA file. To run the RSA algorithm, you can input **java .\RSA.java -help** on the terminal to get extra information on the program and how to use the class to generate key pairs. To generate a key pair, you can input **java -Dprime\_size=<input size> .\RSA.java -gen "<text>"** to run the RSA algorithm. The text input has to be enclosed in quotation marks.

Moving on to part B, we have the Hash and One Time Key files. To run the Hash function, you can input j**ava .\Hash.java <ndatabyte> <ncheckbyte> <pattern> <k> <text>**. The first input, ndatabyte, is an integer input which represents the number of data bytes. The second input, ncheckbytes, is an integer input which represents the number of checksum bytes. The third input, pattern, is an integer input which represents a one byte bit pattern. This input has to be less than 256. The fourth input, k, is an integer input which represents an odd integer. The input has to be odd or the program won’t work. The inputs k and pattern are known to the user and server. The final input, text, is some text that will be the message that will be encoded. The text does not have to be enclosed in quotation marks.

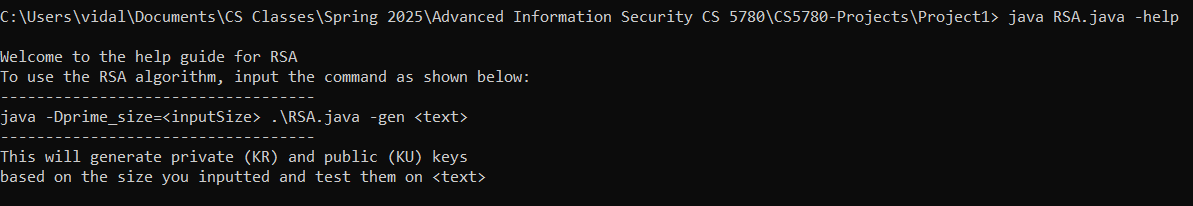
Moving forward, to run the One Time Key algorithm individually, you can input **java .\OneTimeKey.java <key> <text>**. The first input, key, is the one-time key that will be used to encode the text. The key input can be an integer, letters, or a mix of both. The second input, text, is the text that will be encoded by the key. The text can be an integer, a string, or a mix of both but must be perfectly divisible by the key. For example, if the key length is 3, then the text length must be divisible by 3.

Finally, we get to part C, the simplified secure socket layer. To run the secure socket layer, you must open two terminals, preferably on the command prompt. On both terminals, navigate to the directory where the project files are. Once there, one terminal will be the client while the other terminal will be the host. First compile the Java files using javac. After you have compiled the files, on one terminal you input the arguments java SimpleServer 3445 to create the server. On the other terminal, you input the arguments java SimpleClient <XXX> 3445 < <filename>. The input, XXX, is the machine’s host name while the file name is a file within the project directory.

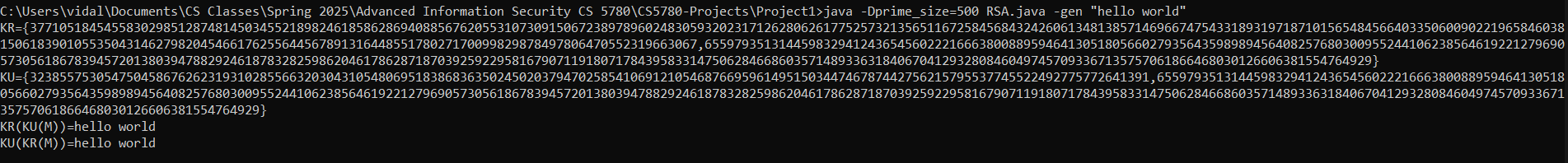
1. Screenshots of each test

Part A

For the input **java RSA.java -help**, we get the following output:

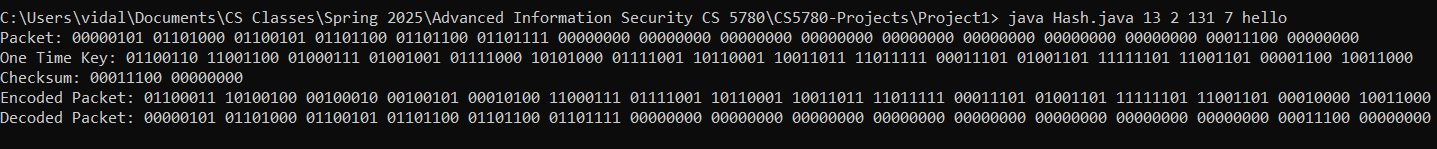


For the input **java -Dprime\_size=500 RSA.java -gen "hello world",** we get the output:

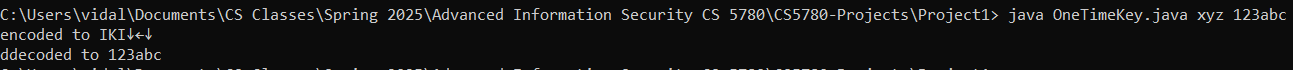


Part B

For the input **java Hash.java 13 2 131 7 hello,** we get the output:

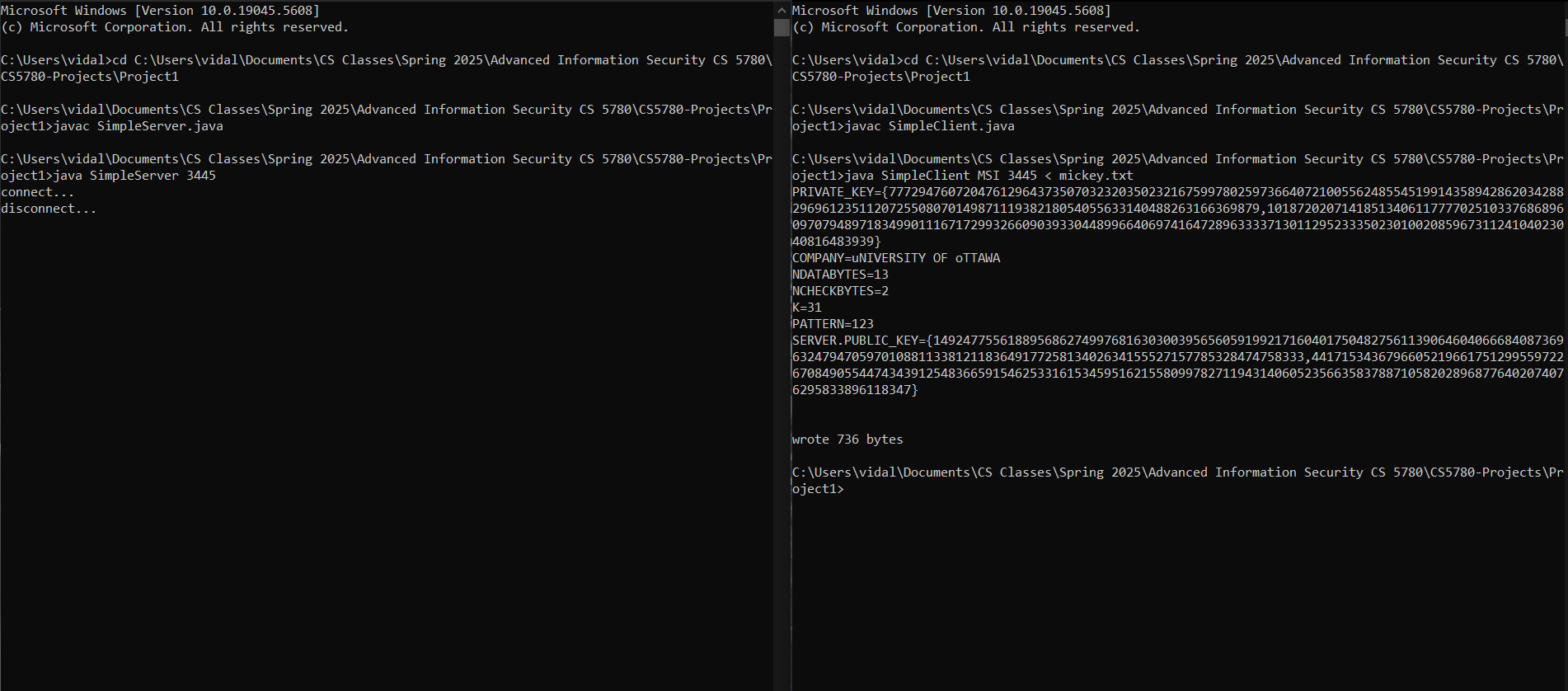


For the input **java OneTimeKey.java xyz 123abc,** we get the output:



Part C

In one terminal, we input **java SimpleServer 3445,** and on another terminal we input **java SimpleClient XXX 3445 < mickey.txt (XXX is the host name of the machine).**

****

1. Percentage of contribution

All members contributed equally to the project