5.3 Simulation 3

This simulation was about performing a man-in-the-middle attack. You are given three programs: a server, a client, and a listener. The server and client think they are communicating with each other, although they are actually communicating via the listener. This simulates the principle of a man-in-the-middle attack where, in this case, the listener is able to intercept all communications between the server and client. An example output from the listener is shown below:

```
listener [from server]: 5
listener [from server]: 3209
listener [from client]: 1
listener [from server]: 4
listener [from server]: Yj76onnI54fIn9ZDeC1rTrRmRNkgNmJlOSHR+LkPNsw=
listener [from client]: K0dRBZh4btBP3YPFUT6yqg==
listener [from client]: 75fe4nKybnWnfVqQE+khgsgMU4H5t2nhK6RkodZ9FvM=
listener [from client]: L6GjqEvxMjfqsra6cjaWbg==
listener [from server]: 7E+tNjo0ZL0PBqzmV2GxT2QxGsI9HuwVZa5JZ9a0rAs=
listener [from client]: iVRtDZFDSUPBGQTKndAynA==
listener [from server]: 4cYJOdSZQYHZnYN1YYwcZepSw/FgzhT6IpcohSGIZ8g=
listener [from client]: L6GjqEvxMjfqsra6cjaWbg==
listener [from server]: J99j76yZEvFjusOO2EmdxT5fNF4iHkCC+KhyRAbRDPM=
listener [from client]: 2O5Ja6BWc7GaV/Uci6k9fg==
listener [from server]: 1bXwZ9jkY9I6hC0F6ptfK6iiGiMnyxKmso82d5co[..]
listener [from client]: vdwh6HgokQF8nakzznTq2w==
listener [from server]: ipVwZxKFpJuqp1iHf+j4Aw==
```

Each time the programs are run, different results are produced.

Step-by-step instructions for how to solve this simulation are provided below. A completed and commented implementation may be found at:

`https://github.com/dcs-cs263/lab3/tree/solutions`

Ex8 Even though you are able to listen in on the communications, the messages between the server and client are encrypted. We can note that:

- The first four messages are always integers. The first two integers are always prime numbers.
- All messages afterwards appear to be Base64 encoded. This can be inferred from the format of the strings. Base64 is used to encode byte data in ASCII strings. You could use a command line tool such as `base64` to quickly decode them or, for example, in Java, this could also be done as follows:

```
Base64.Decoder decoder = Base64.getDecoder();
byte[] bytes = decoder.decode(encodedString);
```

Experimenting with the decoded byte data does not yield any useful results, however. For example, trying to decode it as ASCII or UTF8 strings or interpreting it as images, etc. does not produce meaningful results.

We can assume that the byte data is encrypted. For the data to be encrypted, the server/client must have either a symmetric encryption key or asymmetric encryption key pairs. Based on our observation that the first four messages are always integers and the first two are prime numbers. In the lectures, we have discussed key exchange protocols such as Diffie-Hellman and we know that a Diffie-Hellman-style key exchange begins in precisely this manner. Therefore, it seems logical to try this avenue. As we know from the lectures, Diffie-Hellman-style key exchange may be vulnerable to man-in-the-middle attacks where the man-in-the-middle negotiates keys with the two parties, rather than letting them negotiate keys among themselves. The next step is to modify the listener application so that it performs its own key negotiations with the server and client and no longer passes those messages on to the intended recipients.

Key negotiation with the server is implemented in:

`https://github.com/dcs-cs263/lab3/commit/8f111bc68834d487900ac295ae1b5b935fb494ee`

Key negotiation with the client is implemented in:

`https://github.com/dcs-cs263/lab3/commit/63339e049c006eaefeae5d50c4869c3b230a68cf`

Once we have negotiated our own keys with the server and client, we need to figure out what encryption scheme is used and how the keys are used. Given that we have negotiated symmetric keys with Diffie-Hellman, it would make sense to try a symmetric encryption scheme before moving to anything else. There are many symmetric encryption methods (as we know) and any one of them could be used. Given that the data that is exchanged between the server and client is in the form of byte data, we can assume that the encryption method works on byte data. One such encryption method that we are familiar with from the lectures is AES. It makes sense to first try something that we are familiar with in the context of this exercise.

AES requires keys of specific sizes (*e.g.* 128 bits), however, and keys we have negotiated are not necessarily of the right size. As usual, it is worth trying the simplest method to *pad the key*. This simple, but bad, method is by converting the key into a byte array and padding it with zeros until we reach the right size such as 128 bits (16 bytes):

```
byte[] key = new byte[16];
byte[] kArray = k.toByteArray();
System.out.println(kArray.length);
System.arraycopy(kArray, 0, key, 0, kArray.length);
```

Key padding is implemented in:

`https://github.com/dcs-cs263/lab3/commit/e77b1d1554587259492a72f53cda04465dab14a1`

Since AES is a block cipher, different modes of operation are available[13] to determine how each block is encrypted. It is easy for us to experiment with each of them to see what happens. For example, we can test `AES/CBC/NoPadding` with:

```
private static final String ALGORITHM = "AES/CBC/NoPadding";

public String decrypt(String cipherText) throws Exception
{
  Base64.Decoder decoder = Base64.getDecoder();
  byte[] cipherBytes = decoder.decode(cipherText);
  SecretKeySpec secretKey = new SecretKeySpec(key, ALGORITHM);
  Cipher cipher = Cipher.getInstance(ALGORITHM);
  cipher.init(Cipher.DECRYPT_MODE, secretKey);

  return new String(cipher.doFinal(cipherBytes));
}
```

The data we receive from e.g. the server or client can then be run through this `decrypt` method to see if we get useful data back or if an error occurs. For most modes of operation, we will get an error. After some experimentation, you may determine the exact mode used (it depends on your system – the system default can be selected by just specifying the string `"AES"` rather than a specific configuration). An AES helper class is implemented in:

`https://github.com/dcs-cs263/lab3/commit/b4490bb3282df52dc81b4d8f67b78b8c7ecbc95e`

[13] A list of available options for the Java implementation is available at `https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html`

We can now try to actually communicate with the server and client. From our earlier observations, we note that the server sends a message to the client and the client then follows this up with two messages. When decrypting the message from the server, it initially seems like a random string. However, we can note that it is the string `"o28uyrhkjnkA12iJKHAL"` every time the program is run. So let us try sending this to the client:

```
// initialise our AES helper
InsecureAES aes = new InsecureAES(key);

// send the server's initial message, encrypted
out.println(aes.encrypt("o28uyrhkjnkA12iJKHAL"));
```

This is implemented in:

`https://github.com/dcs-cs263/lab3/commit/93edf7c70cd226b45b2da9ad51ce41ee8ca022a2`

After implementing and running this, we now receive three messages from the client in response:

```
listener [from client]: LordBalaclava
listener [from client]: Mw3JfcBRA0HyylpIQc0vvQ==
listener [from client]: ls
```

It appears that the first message contains a name, the second message contains a Base64-encoded string, and the third appears to be the unix command `ls`. We may assume that the first message is a username, the second is some representation of a password, and the third is a command for the server to execute. We can try sending the same three messages to the actual server. This is implemented in:

`https://github.com/dcs-cs263/lab3/commit/deacdbf60732e5fc839a936f54ed529d982fe48a`

When run, we can see that the server responds with a directory listing. We have successfully taken full control of both connection now and can send arbitrary commands to the server.

Ex9 The second task requires us to steal data from the server. Since we have taken full control of the connection already and it seems to be responding to Unix commands, we can experiment sending other Unix commands to the server. From running the `ls` command we know that there appears to be a `secrets` folder. Let's try sending `cd secrets` followed by `ls` to the server:

```
listener [from server]: changed directory to: secrets
listener [from server]: imc_access_code idl_cctv_locations wmg_server_ips
pitstop_cafe_menu
```

We can then try sending a command to retrieve the contents of one of the files, such as `cat imc_access_code` which causes the server to respond with the file's contents. We have successfully retrieved the secret data from the server. These final changes are implemented in:

`https://github.com/dcs-cs263/lab3/commit/50dc81ff7dfe3886453e31fb92e7e50cecf51e51`