Principles

One of our primary objectives in this course is to make clear the fundamental distinction between functional and nonfunctional characteristics of distributed systems. The functional characteristics describe the business or organizational purpose of the system. The non-functional characteristics affect the quality of the system. Is it fast? Does it easily interoperate with others? Is it reliable and secure?

In this project, we attempt to illustrate one important nonfunctional characteristic of distributed systems – security. Security is a major concern of many distributed system designers.

In Task 1, we will build a secure system using symmetric key cryptography. Alice and Bob have a shared secret and use it to encrypt and decrypt communications. We will encrypt data before writing it to TCP sockets. The Tiny Encryption Algorithm (TEA) will provide the encryption and decryption. User authentication with user ID’s and passwords is done in the application layer – just above the encryption layer. This is a common theme in modern systems. User names and passwords are passed over an encrypted, but otherwise insecure, channel.

In Task 1, the clients and the server all are fitted with the same symmetric key. This would normally require that all parties agree beforehand on the value of the symmetric key. In Task2, we extend our work in Task1. We use asymmetric key cryptography to establish the shared secret. Then, with the symmetric keys in hand, we use symmetric key cryptography as before. We will use RSA for the asymmetric key cryptography. Our implementation of RSA will use the Java BigInteger class and will not make use of any Java crypto API’s.

Be sure to spend some time reflecting on the functional and nonfunctional characteristics of your work. There will be questions on the exams concerning these characteristics. You should be able to demonstrate a nuanced comprehension of course content and be able to explain the technical aspects in relation to potential real world applications.

For each task below, you must submit screenshots that demonstrate your programs running. These screenshots will aid the grader in evaluating your project.
Documenting code is also important. Be sure to provide comments in your code explaining what the code is doing and why.

**Review**

In Project 1 we worked with JEE servlets and Java Server Pages using the Glassfish application server. We worked with mobile device awareness and the Model View Controller design pattern (MVC). We worked with separation of concerns.

In this project we will be working at a lower level. That is, we will not have the Glassfish runtime environment to rely on. You may, however, continue to use Netbeans for all of this work. In this project, we will be programming with TCP sockets. We will not be building web applications. In Netbeans, create projects that are Java Applications – not web applications.

**Task 1**

This Netbeans project will be named Project2Task1.

In this task we will make use of the Tiny Encryption Algorithm (TEA). You are not required to understand the underlying mechanics of TEA. You will need to be able to use it in your code. TEA is one of many symmetric key encryption schemes. TEA is well known because of its small size and speed.

In Figure 4.5 and 4.6 of the Coulouris text, two short programs are presented: TCPClient and TCPServer. You can also find these programs at:


The TCPClient program takes two string arguments: the first is a message to pass and the second is an IP address of the server (e.g. localhost). The server will echo back the message to the client. Before running this example, look closely at how the command line argument list is used. You will need to include localhost on the command line. In Netbeans, command line arguments can be set by choosing Run/Set Project Configuration/Customize.

Make modifications to the TCPClient and TCPServer programs so that spies in the field are able to securely transmit their current location (longitude, latitude, altitude) to Intelligence Headquarters (at a fixed location). Intelligence Headquarters is run by Sean Beggs. There are three spies and one spy commander as listed here:

<table>
<thead>
<tr>
<th>User-id</th>
<th>password</th>
<th>title</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>jamesb</td>
<td>james</td>
<td>spy</td>
<td>long, lat, alt</td>
</tr>
<tr>
<td>joem</td>
<td>joe</td>
<td>spy</td>
<td>long, lat, alt</td>
</tr>
<tr>
<td>mikem</td>
<td>mike</td>
<td>spy</td>
<td>long, lat, alt</td>
</tr>
</tbody>
</table>
The spies are required to inform Sean of their locations as they move about the world (on super secret missions). Sean uses Google Earth Pro to view the locations of his spies. The spies communicate over a channel encrypted using TEA. TEA is a symmetric key encryption algorithm and so Sean has provided his spies with the symmetric key before they left Hamburg Hall on their missions. Of course, Sean’s software knows the ID and password of each spy. So, while TEA is used for encryption, authentication is provided by the user name and password.

Name these two new programs TCPSpyUsingTEAandPasswords.java and TCPSpyCommanderUsingTEAandPasswords.java. The first is a TCP client used by each spy in the field. The second is a TCP server used by Spy Commander Beggs.

Here is an example execution of the server on Sean’s machine.

```
java  TCPSpyCommanderUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
thisissecretsodon’ttellanyone
Waiting for spies to visit...
Got visit 1 from mikel
Got visit 2 from joem
Got visit 3 from jamies
Got visit 4 illegal symmetric key used. This may be an attack.
Got visit 5 from James. Illegal Password attempt. This may be an attack.
```

Here is an example execution of the client on Mike’s machine.

```
java  TCPSpyUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
thisissecretsodon’ttellanyone
Enter your ID: mikem
Enter your Password: mike
```
Enter your location: -79.956264,40.441068,0.00000
Thank you. Your location was securely transmitted to Intelligence Headquarters.

Here is an example execution of the client on Joe’s machine.
java TCPSpyUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
thisissecretsodon’ttellanyone
Enter your ID: joem
Enter your Password: joe
Enter your location: -79.945389,40.444216,0.00000
Thank you. Your location was securely transmitted to Intelligence Headquarters.

Here is an example execution of the client on James Bond’s machine.
java TCPSpyUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
thisissecretsodon’ttellanyone
Enter your ID: jamesb
Enter your Password: james
Enter your location: -79.940450,40.437394,0.00000
Thank you. Your location was securely transmitted to Intelligence Headquarters.

Here is an example execution of the client by Mallory. (She broke into Joe’s office and ran his copy of the client.)
java TCPSpyUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
IBetTheyUseThisKeyAsTheSecret
Enter your ID: joem
Enter your Password: sesame
Enter your location: -79.940450,40.437394,0.00000
An exception is thrown on client. The server ignores this request after detecting the use of a bad symmetric key. It closes the socket and notifies Sean. Your server will detect the bad symmetric key by checking if each character in the decrypted string is ASCII. That is, each character must be found to be less than 128.

Here is an example execution of the client by James Bond (who forgot his password).

```
java  TCPSpyUsingTEAandPasswords
Enter symmetric key for TEA (taking first sixteen bytes):
thisissecretsdonthettellanyone
Enter your ID: jamesb
Enter your Password: jimmy
Enter your location: -79.940450,40.437394,0.00000
Not a valid user-id or password.
```

After each visit by an authenticated spy, the server writes a file called SecretAgents.kml to Sean’s desktop. Here is a copy of a typical KML file. This file can be loaded into Google Earth Pro.
SecretAgents.kml

<?xml version="1.0" encoding="UTF-8" ?>
<kml xmlns="http://earth.google.com/kml/2.2">
  <Document>
    <Style id="style1">
      <IconStyle>
        <Icon>
        </Icon>
      </IconStyle>
    </Style>
    <Placemark>
      <name>seanb</name>
      <description>Spy Commander</description>
      <styleUrl>#style1</styleUrl>
      <Point>
        <coordinates>-79.945289,40.44431,0.00000</coordinates>
      </Point>
    </Placemark>
    <Placemark>
      <name>jamesb</name>
      <description>Spy</description>
      <styleUrl>#style1</styleUrl>
      <Point>
        <coordinates>-79.940450,40.437394,0.0000</coordinates>
      </Point>
    </Placemark>
  </Document>
</kml>
<Placemark>
  <name>joem</name>
  <description>Spy</description>
  <styleUrl>#style1</styleUrl>
  <Point>
    <coordinates>-79.945389,40.444216,0.00000</coordinates>
  </Point>
</Placemark>

<Placemark>
  <name>mikem</name>
  <description>Spy</description>
  <styleUrl>#style1</styleUrl>
  <Point>
    <coordinates>-79.948460,40.444501,0.00000</coordinates>
  </Point>
</Placemark>

When loaded, SecretAgents.kml looks similar to this in Google Earth.
Note that Joe is at work in Hamburg Hall, Mike is hanging out at Starbucks and James Bond is golfing. To see where Sean is located, load the KML file above into Google Earth Pro.

You are required to rewrite the entire file (SecretAgents.kml) after each visit from a spy. The file always contains data on all three spies and the spy commander. This means that you need to maintain the state on the server for each spy. This would include location data, user ID, password, and a title for display on Google Earth Pro (see the Description elements in the KML file).

If a visitor (spy) does not have the correct ID or password, no change will be made to the SecretAgents.kml file. The server will send a message to the client saying illegal ID or password.

If a visitor (evil spy) enters an illegal symmetric key, the server will detect that and close the socket. How the client behaves at this point is of no real concern. The server should not deal at all with anyone with an illegal symmetric key. You will detect the use of this attack by checking if the decrypted data is all ASCII text. It would be a mistake to have the symmetric key stored in the Java code on the client. That is, you
can’t simply test that the user entered symmetric key against a key stored in the client side code.

You may assume that the location data is accurate and well formed. That is, you do not have to validate the longitude, latitude, or altitude. The spies are always careful to enter these data correctly.

Initially, before any spy has communicated with the server using TEA over TCP, all of the spies have their initial state stored in memory objects on the server. How you do this is of your own design. Each spy is initially located in Hamburg Hall (see Joe’s coordinates in the KML file).

As soon as the first spy visits using TEA and TCP, the SecretAgents.kml file is re-written with that spy’s new location. The other values (for the other spies still located in Hamburg Hall) are also re-written to the file. Thus, the file should always have data for all three spies (in Hamburg Hall or not). The KML file only needs to be written and is never read by your program. It is read only by Google Earth Pro. The KML file may be written as a single Java String. There is no need for an XML parser, we are only writing an XML string to a file.

From the Spy Commanders point of view, he runs the server and leaves it running all day and all night. On occasion, perhaps every few hours, he loads the SecretAgents.kml file into Google Earth Pro to see where his spies are located. We are not writing an automatic refresh into Google Earth Pro (maybe next term).

See Wikipedia and see the course schedule for a copy of TEA.java (which you may use.) Name this project Project2Task1. It will contain the files:

TCPSpyUsingTEAandPasswords.java and
TCPSpyCommanderUsingTEAandPasswords.java
TEA.java. Other files may be included as needed.

In my solution, since I am reading and writing streams of bytes, I did not use writeUTF() and readUTF(). Instead, I used these methods in DataInputStream and DataOutputStream:

```java
public final int read(byte[] b)
public void write(byte[] b)
```

The return value of the read method came in very handy.

Note: You may not assume that the symmetric key entered (by a spy) is valid. That is, you should detect when an invalid key is being used. You may assume that the key that the Spy Commander enters is correct and has been secretly provided to
and memorized by each spy. Hint: Postpone this concern until you have the happy case working.

Finally, rather than storing the user id and password in the server side code (insecure if Eve steals a copy of the server source code), store the user id, some cryptographic salt and a hash of the salt plus the password in the code. When authenticating, use the user id to find the salt and hash of salt plus password pair. Hash the salt with the user provided password and check for a match with the stored hash of salt plus password pair. You may use SHA-1 or MD5 as you did in the first project. Hint: Do this last, after everything else is working well. You will need a separate program to help you compute these values. Name your program PasswordHash.java. Include PasswordHash.java in your Task1 project. This class and its execution behavior is in you hands. It should be documented so that the grader can read how it works.

Task 2

The problem with our solution in Task 1 is that the symmetric keys must be agreed to and shared beforehand. Modify Task 1 so that it has the same functional characteristics but uses RSA to establish the shared secret.

Be sure to see RSAExample.java on the course schedule. It contains logic to generate public and private keys. It also shows how we can use those keys to encrypt and decrypt messages. Your solution will contain BigInteger arithmetic to carry out all of the RSA related work. We will not be using any Java cryptography API’s to perform this work.

In order to generate a non-negative session key (RSA does not work with negative integers) use

```java
Random rnd = new Random();
BigInteger key = new BigInteger(16*8,rnd);
```

Name the Task2 programs TCPSpyUsingTEAandPasswordsAndRSA.java and TCPSpyCommanderUsingTEAandPasswordsAndRSA.java. The first is a TCP client used by each spy in the field. The second is a TCP server used by Spy Commander Beggs.

Task 2 Execution

Spy Commander Beggs is the only player with permanent keys. He has generated his own public and private key pair using RSA. He has provided his public keys to each spy (in fact, he has placed his public keys on the Heinz College web site
for everyone to see.) Only Sean knows his private keys. (It is OK if Sean’s private key is hard coded in the server side code.)

When TCPSpyUsingTEAandPasswordsAndRSA is run, it generates a random 16-byte key for TEA. It encrypts this key with Begg’s public key and sends that encrypted key to Sean. (It’s OK to hard code Sean’s public key in the client side code.) Sean decrypts the TEA key with his private key and is able to communicate with the spy as before. Execution of client side code now looks like this:

```bash
java TCPSpyUsingTEAandPasswordsAndRSA
Enter your ID: jamesb
Enter your Password: james
Enter your location: -79.940450,40.437394,0.00000
Thank you. Your location was securely transmitted to Intelligence Headquarters.
```

You may assume that the spies enter data properly. Your program should handle bad passwords and ID pairs with an appropriate error message - as in Task 1.

Final note. It is a bad idea to write your own cryptographic software – unless you are an expert with years of experience. We do this exercise only to understand and explore some major issues in computer security.

## Questions

Questions should be posted to the Blackboard Discussion Board, under the Project 2 Discussion Forum.

## Project 2 Summary

Be sure to review the grading rubric on the schedule. We will use that rubric in evaluating this project. Documentation is always required. Remember to separate concerns.

The Netbeans projects will be named as follows:

- Project2Task1  (You need to zip this folder)
- Project2Task2  (You need to zip this folder)

You should also have two screen shot folders:

- Project2Task1-screenshot  (Do not zip)
- Project2Task2-screenshot  (Do not zip)
For each Netbeans project, use “File->Export Project->To Zip”. You must export in this way and NOT just zip the Netbeans project folders. In addition, zip all the Netbeans export zips and the screenshot folders into a folder named with your andrew id.

Zip that folder and submit it to Blackboard.

The submission should be a single zip file. This file will be called YourAndrewID.zip.

Submission file structure:

YourAndrewID.zip

- Project2Task1.zip
- Project2Task2.zip
- Project2Task1-screenshot
- Project2Task2-screenshot