95-702 Distributed Systems

Project 2

Assigned: Friday, Feb-9, 2018

Due: Due Friday Feb-23, 11:59pm

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 are all fitted with the same symmetric key. In other words, 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. In Task 2, our implementation of RSA will use the Java BigInteger class and will not make use of any Java crypto API's. In Task3, we will use Java's keytool to generate the RSA keys for us.

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.

Separate concerns. Your code should be modular and each module should do one thing well.

Review and Project 2 Introduction

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.

In Project 2, we will pretend that the City of Pittsburgh is embarking on a new smart city initiative. As one small part of that initiative, our company, GunShotSensing Inc. has been hired to install our new audio sensors in high crime areas of the city. Often placed indoors and due to hardware constraints (short battery life and no GPS receiver), we need to manually report on each sensor's location. These sensors ignore most sounds but are able to detect gunshots. When gunshots are detected, a JSON message containing the sensor id is transmitted through the cellular network and then over the internet to our headquarters. When we receive the message, we contact the local police. We are able to report three things to the authorities. One, our sensor has detected gunshots. Two, we know the longitude and latitude of the sensor. And three, we can view the sensor's location on Google Earth Pro – helping to guide the police to the precise location.

In Project 2, we will be implementing the client and server software involved with sensor installation. We will not be handling the sensor reports of gun fire.

Task 1 50 Points.

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:

http://www.cdk5.net/wp/extra-material/source-code-for-programs-in-the-book

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.

Spend some time studying and experimenting with TCPClient and TCPServer. Become familiar with socket programming,

We have three sensor installers. Each of these installers has been provided with our symmetric key. The installation locations are determined by the good judgement of our installers – highly skilled in the hiding of sensors in high crime areas. After an installer selects a location and installs a sensor, his or her computer transmits an encrypted JSON string in the following format:

{“ID”: “user-id”,

”passwd”: ”password”,

“Sensor ID”: 6-digit integer,

“Latitude”: latValue,

“Longitude”: longValue

}

Be sure to understand the JSON format as described at www.json.org.

On occasion, an installer will move a sensor. In that case, the installer sends a JSON message using the same sensor ID as one that has already been installed. The server will simply replace the old location of that sensor with the new location. We require that the sensor ID and the sensor’s location be maintained in a Java TreeMap on the server.

At GunShotSensing Inc. headquarters, we need a real time update on each sensor’s location. Our installers send in these updates immediately after initial installation or after a movement. When gun shots are reported, Google Earth is used to view all of the sensor locations. The sensor’s ID is used to find the sensor on Google Earth Pro. Police may then be told of the location of the gun shots. Sensors being moved are turned off while in transit.

Make modifications to the TCPClient and TCPServer programs so that installers in the field are able to securely transmit sensor location data. Identification, authentication, authorization and privacy are all of primary concern. Only Chief Sensor Installers are permitted to move a sensor and report on its new location. We have assigned each installer an ID and password. Here is a list of our installers and their passwords. (We are revealing passwords here but passwords are normally only known by the installers).

User-id	Password	Title
Moe	moeh	Chief Sensor Installer
Larry	larryh	Associate Sensor Installer
Shemp	shemph	Associate Sensor Installer

At GunShotSensing Inc. headquarters, Google Earth Pro is used to view the locations of sensors. Google Earth Pro reads a KML file in order to render a map showing sensor locations. Our server software creates a new KML file (replacing the old KML file) after each sensor installation or movement. We store a file named Sensors.kml on our desktop. Hint: In Java, to find the desktop, use

```
String deskTopLocation = System.getProperty("user.home") +
"/Desktop/Sensors.kml";
```

The three installers communicate over a channel encrypted using TEA. TEA is a symmetric key encryption algorithm and so we have provided each installer with the symmetric key before they begin their installation runs. Of course, our server software knows the symmetric key as well as the ID and password of each installer. So, while TEA is used for encryption, identification and authentication is provided by the user name and password. Authorization is provided by determining if the current user is a Chief Sensor Installer.

Name these two new programs TCPInstallerUsingTEAandPasswords.java and TCPGunShotIncUsingTEAandPasswords.java. The first is a TCP client used by each installer in the field. The second is a TCP server used by the managers at GunShots.

Here is an example execution of the server:

```
java TCPGunShotIncUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
1234567890123456
Waiting for installers to visit...
Got visit 1 from Moe, Chief Sensor Installer
```

Got visit 2 from Larry, Associate Sensor Installer

Got visit 3 from Moe, Chief Sensor Installer, a sensor has been moved.

Got visit 4 from Moe, Chief Sensor Installer

Got visit 5 illegal symmetric key used. This may be an attack.

Got visit 6 from Shemp, Associate Sensor Installer

Illegal Password attempt. This may be an attack.

Here is an example execution of the client on Moe's machine.

```
java  TCPInstallerUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
1234567890123456
Enter your ID: Moe
Enter your Password: moeh
Enter sensor ID: 012379
Enter new sensor location: -79.956264,40.441068,0.00000
Thank you. The sensor's location was securely transmitted to GunshotSensing
Inc.
```

Here is an example execution of the client on Larry's machine.

```
java  TCPInstallerUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
1234567890123456
Enter your ID: Larry
Enter your Password: larryh
Enter sensor ID: 876021
Enter new sensor location: -79.945389,40.444216,0.00000
Thank you. The sensor's location was securely transmitted to GunshotSensing
Inc.
```

Here is another example execution of the client on Moe's machine.

```
java  TCPInstallerUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
```

1234567890123456

Enter your ID: Moe

Enter your Password: moeh

Enter sensor ID: 012379

Enter new sensor location: -79.952367,40.441187,0.00000

Thank you. The sensor's new location was securely transmitted to GunshotSensing Inc.

Here is a third example execution of the client from Moe's machine.

java TCPInstallerUsingTEAandPasswords

Enter symmetric key as a 16-digit integer.

1234567890123456

Enter your ID: Moe

Enter your Password: moeh

Enter sensor ID: 012380

Enter new sensor location: -79.955264,40.441568,0.00000

Thank you. The sensor's location was securely transmitted to GunshotSensing Inc.

Here is an example execution of the client by Mallory. (She stole Shemp's machine and ran his copy of the client.)

```
java  TCPInstallerUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
1934547890123453
Enter your ID: Shemp
Enter your Password: sesame
Enter sensor ID: 012387
Enter new sensor location: -79.955264,40.441568,0.00000
```

An exception is thrown on the client. The server ignores this request after detecting the use of a bad symmetric key. It closes the socket and notifies the staff at Gunshot Inc. Your server will detect the bad symmetric key by checking if each character in the decrypted string is ASII. That is, each character must be found to be less than 128.

Here is an example execution of the client by Shemp (he got his machine back from Mallory but forgot his password).

```
java  TCPInstallerUsingTEAandPasswords
Enter symmetric key as a 16-digit integer.
1234567890123456
Enter your ID: Shemp
Enter your Password: Shemph
Enter sensor ID: 345671
Enter new sensor location: -79.958964,40.423568,0.00000
Illegal ID or Password
```

After each visit by an authenticated installer, the server writes a file called Sensors.kml. Here is a copy of a typical KML file. This file can be loaded into Google Earth Pro.

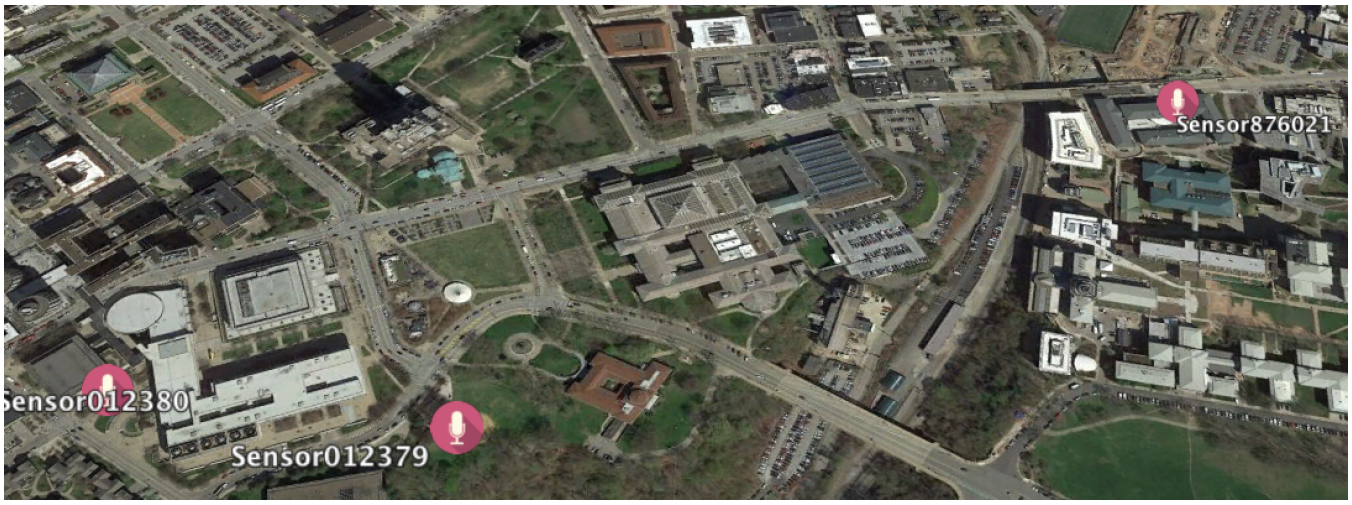
Note the way that the installer's name is placed within the description tag. This is easy to view by clicking directly on the microphone icon in Google Earth Pro.

Sensors.kml

```
<?xml version="1.0" encoding="UTF-8" ?>
<kml xmlns="http://earth.google.com/kml/2.2">
<Document>
  <Style id="style1">
    <IconStyle>
      <Icon>
        <href>https://lh3.googleusercontent.com/MSOuW3ZjC7uflJAMst-
cykSOEOwl_cVz96s2rtWTN4-Vu1NOBw80pTqrTe06R_AMfxS2=w170</href>
      </Icon>
    </IconStyle>
  </Style>
<Placemark>
  <name>12379</name>
  <description>Moe</description>
  <styleUrl>#style1</styleUrl>
  <Point>
    <coordinates>-79.952367,40.441187,0.00</coordinates>
  </Point>
</Placemark>
<Placemark>
  <name>12380</name>
  <description>Moe</description>
  <styleUrl>#style1</styleUrl>
  <Point>
    <coordinates>-79.955264,40.441568,0.00</coordinates>
  </Point>
</Placemark>
<Placemark>
```

```
    <name>876021</name>
    <description>Larry</description>
    <styleUrl>#style1</styleUrl>
    <Point>
        <coordinates>-79.945389,40.444216,0.00</coordinates>
    </Point>
</Placemark>
</Document>
</kml>
```

When loaded, Sensors.kml looks similar to this in Google Earth.



You are required to rewrite the entire file (Sensors.kml) after each visit from an authenticated and authorized installer. The file always contains data on all deployed sensors. This means that you need to maintain the state on the server holding the locations of each sensor and who last installed it. This would include the sensor ID and sensor location and the installer's name. You are required to use a Java TreeMap to hold this data.

Note that Larry installed a sensor on top of Hamburg Hall. Your execution in Task 1will show that, in addition, Shemp stored a new sensor across the street in the new Tepper Quad area. (You will need to figure out the longitude and latitude of this new installation).

Take a screen shot of Google Earth Pro after the interactions above are complete (be sure to show Shemp's installation at Tepper). In addition, submit a copy of the KML file that is generated after these interactions.

If a visitor (installer) does not have the correct ID or password, no change will be made to the KML file. The server will send an encrypted message to the client saying "illegal ID or password".

If a visitor (trying to pretend to be an installer) enters an illegal symmetric key, the server will detect that and close the socket. The server should not deal at all with anyone with an illegal symmetric key. On the server, 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. On the client, the user should be informed that the server has closed the communications channel.

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 installers are always careful to enter these data correctly.

Initially, before any installer has communicated with the server using TEA over TCP, the collection (TreeMap) holding the sensor data is empty. There is no KML file. The KML file is written after the first sensor installation. It is re-written, in its entirety, after each subsequent sensor installation or movement.

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 server operator's point of view, the server is run and left running all day and all night. On occasion, when a shooting report arrives, the KML file is loaded into Google Earth Pro to see where the shots were heard. 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:

TCPGunShotIncUsingTEAandPasswords.java and

TCPInstallerUsingTEAandPasswords.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:

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 an installer) is valid. That is, you should detect when an invalid key is being used. You may assume that the key that the server operator enters is correct and has been secretly provided to and memorized by each installer. 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 (a random integer) 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 are required to use SHA-256 for hashing. Hint: Do this password hash last, after everything else is working well. You will need a separate program to help you compute these values. Name this separate program PasswordHash.java. Include PasswordHash.java in your Task1 project.

PasswordHash.java will have the following user interaction. Note, lines generated from the program are prefixed with a octothorp ('#') character.

#Enter user ID

Moe

#Enter password

moeh

#Generating a random number for salt using SecureRandom

#User ID = Moe

#Salt = e78ace137abf9bd2cfb92a87f0d95ee8888912b4

#Hash of salt + password =

030145629418bacba728f555022add79a95cd793291a73193

bebcf0a12a04dd5

#Enter user ID for authentication testing

Moe

#Enter password for authentication testing

moeh

Validated user id and password pair

Here is another execution of PasswordHash.java:

#Enter user ID

Moe

#Enter password

moeh

#Generating a random number for salt using SecureRandom

#User ID = Moe

#Salt = c1ab548c72752ef5e77caed71c2849aa4a63fccd

#Hash of salt + password =

07c73efb5d7be7a1c8d866e771a7d6ed164161535252e1163

3cdfad94dccf3c7

#Enter user ID for authentication testing

Moe

#Enter password for authentication testing

moey

Not able to validate this user id, password pair.

Note that PasswordHash.java is a utility program, run on its own, as a helpful tool for the Gunshot server administrators.

Task 1 50 Points Rubric

System built according to specifications: -0..-15

Documentation and screenshots : -0..-2.5

Style: Separation of concerns: -0..-2.5

Submission requirements: -0..-2.5

Task 2 40 Points

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. In this task, 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

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

Name the Task2 programs TCPInstallerUsingTEAandPasswordsAndRSA.java and TCPGunShotIncUsingTEAandPasswordsAndRSA.java. The first is a TCP client used by each installer in the field. The second is a TCP server used by GunshotSensing Inc.

The server is the only player with permanent keys. We need to generate the server's public and private RSA key pair. We need to provide the public key to each sensor installer (in fact, we could place the public key material on GunshotSensing's web site for everyone to see.) Only the server at GunshotSensing knows its private keys. (It is OK if the private key is hard coded in the server side code.)

When TCPInstallerUsingTEAandPasswordsAndRSA is run, it generates a random 16-byte key for TEA. It encrypts this key with the server's public key and sends that encrypted key to the server. (It's OK to hard code the server's public key in the client side code.) The server decrypts the TEA key with its private key and is able to communicate with the installer as in Task 1. Execution of client side code now looks like this:

```
java  TCPInstallerUsingTEAandPasswordsAndRSA
Enter your ID: Shemp
Enter your Password: shemph
Enter sensor ID: 345671
Enter new sensor location: -79.958964,40.423568,0.00000
Thank you. The sensor's location was securely transmitted to GunshotSensing
Inc.
```

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

Here is a test case for you to run. Begin with a new run of your server and have the clients behave as follows:

Activities of the sensor installers:

	Sensor ID	Longitude	Latitude	Altitude
Moe installs sensor 1		-79.94893572,	40.44583796,	0.00
Moe installs sensor 2		-79.94893572,	40.44583796,	0.00
Shemp installs sensor 3		-79.96002024,	40.43929865,	0.000
Moe installs sensor 4		-79.95920304,	40.4398469,	0.00
Larry installs sensor 5		-79.95102196,	40.45055952,	0.00
Larry installs sensor 6		-79.95102196,	40.45055952,	0.000
Moe installs sensor 7		-79.94343405,	40.45192574,	0.000
Shemp installs sensor 8		-79.94343405,	40.45192574,	0.000

Moe moves sensor 3 down to Craft avenue -79.96075434, 40.43564356,0.00

Moe moves sensor 8 down to Devonshire -79.94633024, 40.45362031, 0.000

Moe moves sensor 2 to Bigelow -79.95404587, 40.453163, 0.000

Shemp tries to move sensor 5 but is rejected by the server as not authorized

Take a screen shot of Google Earth Pro after these interactions are complete. In addition, submit a copy of the KML file that is generated after these interactions.

Task 2 40 Points Rubric

System built according to specifications: -0..-15 (including the Google Earth Pro screen shot after the interaction shown above with 8 sensors installed. KML file too.)

Documentation and screenshots: -0..-2.5

Style: Separation of concerns: -0..-2.5

Submission requirements: -0..-2.5

Task 3 10 Points + 5 Bonus Points

This task has the same functional and non-functional behavior as Task 2. However, in Task 2 we generated the RSA keys ourselves. This is great for educational purposes but it is not how it is done in the industry. In this task, you will generate the RSA keys with Java's keytool command.

The server side code will need to read the private key data from a keystore file created by Java's keytool command. The client side code will need to read the public

key data from an X509 certificate. This file will also be created by Java's keytool command.

In order to complete this task, look over the SSL lab (Lab5 Security) at http://www.andrew.cmu.edu/course/95-702/Labs/Lab5SSL.txt. But note: WE ARE NOT USING SSL SOCKETS in this project - we are only using the keytool program to create keys and certificates. Use Java's keytool command to create a pair of keys for the server. Name the keystore file GunshotSensorKeys.jks. Use keytool to create a certificate file named GunshotSensorCert.cer. The server will read the private keys from the keystore and the client will read the public key from the certificate file. Execution will then proceed as in Task 2. You will need to do a bit of research to figure out how to read the keystore and certificate files. But here is some help.

This code is found on my server:

```
FileInputStream fis = new FileInputStream(keyFileName);
KeyStore keyStore = KeyStore.getInstance("JKS");
keyStore.load(fis, passWord);
fis.close();
Certificate cert = keyStore.getCertificate(alias);
RSAPublicKey publicKey = (RSAPublicKey)cert.getPublicKey();
Key key = keyStore.getKey(alias,passWord);
RSAPrivateKey privateKey = (RSAPrivateKey)key;
```

This code is found on my client:

```
FileInputStream certFile = new FileInputStream(keyFileName);
BufferedInputStream ksbufin = new BufferedInputStream(certFile);
X509Certificate cert = (X509Certificate)
    CertificateFactory.getInstance("X.509").generateCertificate(ksbufin);
// read the public key
RSAPublicKey publicKey = (RSAPublicKey)cert.getPublicKey();
```

Important 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. Or, perhaps, to spark your interest in this area.

Questions

Questions should be posted to the Piazza Discussion Forum.

Project 2 Summary

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 of 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.

The Netbeans projects will be named as follows:

- Project2Task1 (You need to zip this folder)
- Project2Task2 (You need to zip this folder)
- Project2Task3 (You need to zip this folder)

You should also have three screen shot folders:

- Project2Task1-screenshot (Do not zip)
- Project2Task2-screenshot (Do not zip)
- Project2Task3-screenshot (Do not zip)

Submission file structure:

YourAndrewID.zip contains

- Project2Task1.zip
- Project2Task2.zip
- Project2Task3.zip
- Project2Task1-screenshot
- Project2Task2-screenshot
- Project2Task3-screenshot