Concepts of Parallel and Distributed Systems CSCI-251

rev. 3.3

Project 3: Secure Messaging

1 Goals

Security is important in the age of shared information. We will be looking to use a network protocols to be able to send secure messages to other people and decode messages sent to you

2 Overview

Write a program to use public key encryption to send secure messages to other users. This will be a distributed system where keys will be stored on a server and you will be able to secure messages to classmates using only their email address. You will be required to write the client to encode and decode messages, and the messages will all need to be short (smaller than the key length), in order to reduce the complexity of message encoding/decoding.

3 RSA

You will need to use the prime number generation from your previous project to be able to generate RSA keys (public and private), and use these keys to send secure messages to classmates.

The basic algorithm for RSA is well known and in your lecture notes. You will need to generate a public and a private key when the user runs your program to generate keys.

To generate these keys, you will need a p and q values, which are both prime. Remember, that if your key size is 1024 bits, p and q total bits need to add up to 1024. If p and q are both be 512 bits, when you multiply them, their new size is 1024 bits. For this project, you should NOT make them the same size (1/2 the total number of bits), rather you should pick p to be 1/2 the total number of bits +/- 20-30%. Then select q to be total size-len(p).

The basic algorithm is as follows:

```
N = p * q
r = (p - 1) * (q - 1)
E = a prime number
D = modInverse(E, r)
```

E is often picked as a relatively small prime number (2^{16}) as it makes decryption faster, and no less secure, since you need both e and n to make a decryption key.

In order to generate these keys, you will need a modInverse Function, which is provided for you:

```
static BigInteger modInverse(BigInteger a, BigInteger n)
{
    BigInteger i = n, v = 0, d = 1;
    while (a>0) {
        BigInteger t = i/a, x = a;
        a = i % x;
        i = x;
        x = d;
        d = v - t*x;
        v = x;
    }
    v %= n;
    if (v<0) v = (v+n)%n;
    return v;
}
```

The actual encryption algorithm is given in the lecture notes, but I recommend you use the BigInteger.ModPow function to perform the math, if you don't your code will fail.

The only other challenging part is writing the data. This is going to be Base64 encoded. We use Base64 because it doesn't have any extended ASCII characters, which means it can be cleanly written and read to the console and files on the system. There are built in methods for converting to Base64 strings. You should use these.

4 The Key

The format of the files you write needs to be a standard format so that the keys and messages stored on the server can be used by anyone. Recall that E and D may be any length, so we need a way to keep track of how much of the key is the E, N, and D values.

In the public key, the format is as follows:

`eeeeEEEEEEE....EEEnnnnNNNNNNN...NNN`

e = 4 bytes for the size of E E = the bytes for E n = 4 bytes fo the size of N N = the bytes for N

As an example, if we have: 0002AB0004WXYZ then the 0002 represents two bytes for the E value (AB), after than we have 0004, representing 4 bytes for N, WXYZ.

The same is true for the private key, except d/D replace e/E

Finally, with this particular project the e and n are stored as big endian, with the least significant byte to the left; while E/D and N are stored as little endian. Your keys should be stored the same way.

Once you have constructed the byte array for your key, you should base64 encode it for storage. Note that this occurs AFTER you have the complete byte array.

When we get a key from the server, we perform the following steps:

1. Read the JSON Object

2. Extract the base64 key as a byte array, this is not human readable.
3. Read the first 4 bytes (remember these are big endian)
4. Convert those bytes to a Int named e
5. Skip the first 4 bytes, Read e number of Bytes as E (little endian)
6. Convert E to a BigInteger
7. Skip 4+e bytes, read 4 bytes as n, Check the Endianess
8. Convert n to an Int (big endian)
9. Skip 4+e+4 bytes, read n Bytes into N (little endian)
   You now have E and N which you can use to encrypt a message
   The same logic applies for decoding a private key as well.
   To construct a key, follow the steps the other way around (basically)

5 Requirements

1. Your program must be a .net core command line program:

```
dotnet run <option> <other arguments>
```

   There are several options you can provide as the first argument, they are:
   - keyGen
   - sendKey
   - getKey
   - sendMsg
   - getMsg

   Each of these will accomplish a basic task, as detailed below (along with the the extra command line options)
   - **keyGen** *keysize* - this will generate a keypair of size *keysize* bits (public and private keys) and store them locally on the disk (in files called *public.key* and *private.key* respectively), in the current directory. The format of the private key is:

```
{
  email: <list of emails>,
  key: <base64 encoded key>
}
```

     while the public key is:

```
{
  email: <single email>,
  key: <base64 encoded key>
}
```

     You should also create classes to store these objects in your program.
     When you first generate these keys, there is no email address stored in the email field, that will be added to the list of emails in the private key when sendKey is called. The local public.key will never have an email added to it (it will be added when sent to the server, but not written locally), but should be in the format above.
   - **sendKey** *email* - this option sends the public key that was generated in the keyGen phase and send it to the server, with the email address given. This should be your email address. The server will then register this email address as a valid receiver of messages. The private key will remain locally, though the email address that was

given should be added to the private key for later validation. If the server already has a key for this user, it will be overwritten.

- **getKey** *email* - this will retrieve public key for a particular user (not usually yourself). You will need this key to encode messages for a particular user. You should store it on the local filesystem as <email>.key. (ie, if I issue *getKey jsb@cs.rit.edu* it should write it locally as *jsb@cs.rit.edu.key* so that you can have multiple keys stored and not overwrite your local keys. You should store this key the same way you store your own public key. This is convinient as this is the format you get from the server.
- **sendMsg** *email plaintext* - this will take a plaintext message, encrypt it using the public key of the person you are sending it to, based on their email address. It will base64 encode the message it before sending it to the server (see the server section for the format of the message to send to the server). If you do not have a public key for that particular user, you should show an error message indicating that the key must be downloaded first. The steps to encrypt a message are as follows:
  (a) Ensure you have the public key for the user you are sending a message to, if not, abort
  (b) Take the plaintext message and covert it to a byte array
  (c) Take the resulting byte array and load it into a big integer
  (d) Perform the encryption algorith
  (e) Convert the results big integer to a byte array
  (f) Base64 encode the byte array
  (g) Load the base64 encoded byte array and the email message into message object and send it to the server
- **getMsg** *email* - this will retrieve a message for a particular user, while it is possible to download messages for any user, you will only be able to decode messages for which you have the private key. If you download messages for which you don't have the private key, you should simply state that those messages can't be decoded. You should decode and print anything for which you have the private key. The format of this message is listed in the server section. You should work with the message in the following manner:
  (a) Validate that you have a private key for the email being requested, if not, abort.
  (b) Load the JSON object from the server into a local object
  (c) Base64 decode the content property of the message object into a byte array
  (d) Convert the byte array to a big integer
  (e) Perform the decryption algorithm
  (f) Convert the results big integer to a byte array
  (g) Concert the byte a string
  (h) Display the message

2. If the user specifies invalid command line arguments, you must print out an error message indicating the problem.

6 Server

The server will be listening on **kayrun.cs.rit.edu**, port 5000. The server is listening for TCP/HTTP connections and is setup as a WEB API host. All data is expected to be JSON encoded, and all return messages will be JSON encoded (or no message), and include an HTTP return code.

Common HTTP return codes include:
2xx - OK
3xx - Redirect
4xx - Not found
5xx - Error

If you use the `https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-5.0` class to make your request, you will be able to look at the status code that is being returned to ensure it's in the 200-299 range for success (or use the isSuccessStatusCode).

This class will also allow you to do the required methods (Get and Put) that the server requires to process the request.

For each of the endpoints, there are 2 key actions, Get and Put. Get will be used to get keys and messages while Put is used to Put keys and messages. If I wanted to make a request to get a message for a particular user, I'd send the following request to the server:
GET http://kayrun.cs.rit.edu:5000/Message/user@foo.com
where GET is the type of HTTP request (see the HTTPClient methods) and the URL is in the Server/Action/email format. Actions can be one of the following:

- Message
- Key

which makes possible URLs:

- GET http://kayrun.cs.rit.edu:5000/Message/email
- PUT http://kayrun.cs.rit.edu:5000/Message/email
- GET http://kayrun.cs.rit.edu:5000/Key/email
- PUT http://kayrun.cs.rit.edu:5000/Key/email

Put messages require a JSON body, the data that you are doing to update the database with. For a Key, the message will be in the format:

```
{
  email: <email>,
  key: <base64 encoded key>
}
```

while messages will be in the format:

```
{
  email: <email>,
  content: <base64 encoded message>
}
```

In both cases, the email address sent in the URL MUST match the URL in the request. If they do not, the message will be considered invalid.

Currently, one key exists on the server, *jsb@cs.rit.edu*. You can make a GET request to `http://kayrun.cs.rit.edu:5000/Key/jsb@cs.rit.edu` to see a properly base64 encoded key and the format of both the messages you'll get back and the messages you'll send.

When you want to send JSON encoded messages to the server, you will need to set the content type as JSON, per the following:

```
var content = new StringContent(<serialized json object>,
        Encoding.UTF8, "application/json");
```

this then becomes the content of the request you make to the server.

The server is dumb in that it only returns one message for a user and can only keeps one key for a user. There is no authentication of any sort, so it is possible that you can overwrite someone else's keys and messages. Be NICE! (This will be adjusted shortly to allow returning of all messages)

Additionally, all due diligence has been made to ensure the server will not crash, but it often will not provide useful error messages. If the server is not responding with anything you'd expect, be sure your client is sending the correct data.

7 Design

1. The program must be command line driven
2. The output (minus error messages), must match the writeup
3. Command line help must be provided
4. The program must be designed using object oriented design principles as appropriate.
5. The program must make use of reusable software components as appropriate.
6. Each class and interface must include a comment describing the overall class or interface.

8 Submission Requirements

Zip up your solution in a file called project3.zip. The zip file should contain at least the following files, with the csproj being in the root of the zip.

- Any source files (*.cs)
- Messenger.csproj

I will be testing your program on either a Mac or Windows machine, with secondary tests being run on Linux. You should ensure your program works on multiple platforms.

9 Sample Runs

This assumes I already have public and private keys for jsb@cs.rit.edu

```
dotnet run keyGen test@cs.rit.edu

dotnet run sendKey test@cs.rit.edu
Key saved

dotnet run sendMsg jsb@cs.rit.edu "It worked!"
Key does not exist for jsb@cs.rit.edu

dotnet run getKey jsb@cs.rit.edu
```

```
dotnet run sendMsg jsb@cs.rit.edu "Project 3"
Message written

dotnet run getMsg jsb@cs.rit.edu
Project 3
```

10 Where to start

The order of operations I recommend to complete is as follows:

1. GetKey
   (a) Retrieve jsb@cs.rit.edu from the server using an HTTP Get request
   (b) Write the key to the disk in the current directory (do not specify a path)
2. Process the written key
   (a) Open up the file that you wrote in step 1
   (b) Deserialize it into a a public key object using the Json Deserilize from the Intermediate Networking lecture.
   (c) Base64 Decode the key part of the object
   (d) Split the byte array into nBytes, NBytes, eBytes, EBytes
   (e) Concert the smaller byte arrays to number
   (f) You now know the proper format for a key and can construct your own key.
3. From Here, you can write genKey, SendKey, getMsg, sendMsg in that order.

11 Grading Guide

Your project is out of 100 points. I will grade your project on the following criteria:

- Functionality (80 points)
  - (20 points) keyGen - Generating compatible private/public keys and storing them as private.key / public.key. Keys not base64 encoded and not compatible with the server will earn 0 points.
  - (10 points) sendKey - Ensures there is a public and private key on the local system and sends the public key to the server, using the given email address
  - (10 points) getKey - Retrieves the public key from the server for a given email address and writes it to the file system, base64 encoded.
  - (20 points) sendMsg - Send a message to the server view the correct command line message. Message must be encrypted before sending it to the server
  - (20 points) getMsg - Get an encrypted message from the server and decode it, using the private key, and outputting it the console.
- Style (10 points) - Correct documentation for public methods, name in source
- Misc Design/Usage (10 points) - Correct command line help, OO design

Programs that crash during any of the top level items will automatically earn a 0, even if a subset of the tests pass (for example, if I try to send a key that doesn't exist, but your program works for when a key exists, sendKey will earn a 0)

I will grade the test cases based solely on whether your program produces the correct output as specified in the above Software Requirements. Any deviation from the requirements will result in a grade of 0 for the test case. This includes errors in the formatting (such as missing or extra spaces), incorrect uppercase/lowercase, output lines not terminated with a newline, extra newline(s) in the output, and extraneous output not called for in the requirements. The requirements state exactly what the output is supposed to be, and there is no excuse for outputting anything different. If any requirement is unclear, please ask for clarification.