Concepts of Parallel and Distributed Systems
CSCI-251

rev. 2.2

Project 3: Secure Messaging

Due: Friday, April 17, 2020 at 11:59pm

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, so p and q can both be 512 bits, or one could be 508 bits and the other 516, etc. As long as their total size when multiplied is your key size, it's okay.

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 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 File Format

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 pubic 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, on some systems, the byte array is read backwards (little Endian). You should check if the system is littleEndian and reverse the array when you read the keys off the disk. Note, you should do this without hardcoding a Endian, as different systems may read values differently.

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 (public and private keys) and store them locally on the disk (in files called *public.key* and *private.key* respectively), as base64 encoded keys. Note, it is NOT associated with an email address until it is sent to the server.
- **sendKey** *email* - this option sends the public key and to the server. The server will then register this email address as a valid receiver of messages. The private key will remain locally. If the server already has a key for this user, it will be overwritten. SendKey should also update the local system to register the email address as valid (one for which messages can be decoded). It is recommended to store this with the private key.
- **getKey** *email* - this will retrieve a base64 encoded 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 jeremy.brown@rit.edu* it should write it locally as *jeremy.brown@rit.edu.key* so that you can have multiple keys stored and not overwrite your local keys.
- **sendMsg** *email plaintext* - this will base64 encode a message for a user in the to field. 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.
- **getMsg** *email* - this will retrieve the base64 encoded 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.

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=netcore-3.0` 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, *jeremy.brown@rit.edu*. You can make a GET request to `http://kayrun.cs.rit.edu:5000/Key/jeremy.brown@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(jsonObject.ToString(),
        Encoding.UTF8, "application/json");
```

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:

- 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

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

dotnet run sendKey jeremy.brown@rit.edu
Key saved

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

dotnet getKey jeremy.brown@rit.edu

dotnet run sendMsg jeremy.brown@rit.edu "It worked!"
Message written

dotnet run getMsg jeremy.brown@rit.edu
It worked!
```

10 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 warn 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.