CS 490: Introduction to Cybersecurity(Fall 2020)

Homework 3: RSA Encryption

Due Date: **Tuesday, October 6, 2020, 11:59:59 PM**

1 Background

This assignment will focus on the implementation of the RSA algorithm. Specifically, you will have to implement key generation, encryption and decryption, signing and signature checking, and cracking of RSA messages.

The intent is to implement this in Java, since the JDK provides the functionality to handle the necessary math. You must speak to me first if you want to use another language!

As Java is meant to be cross-platform, there is no specific reference platform for this assignment.

2 Resources

These links are all described below, but are included here, all in one place.

- `RSA.java` starter code, which you *MUST* use.
- Javadoc documentation for the `CipherText`, `RSAKey`, and `RSA` classes (to view the Javadocs, unzip the folder and open the index.html file).
- Command Line Parameter Documentation—note that the provided `RSA.java` file calls the correct functions based on the parameters passed, and you should not change any of this
- The encryption lectures and the Wikipedia article on RSA
- Java SDK documentation, specifically the `java.math.BigInteger` and `java.security.MessageDigest` classes
- The `test-rsa.sh` shell script

3 Assignment

3.1 Prerequisites to Review

You should be familiar with how the RSA algorithm works from the Week04 Primality Lectures and the Week05 RSA Lectures. More details are available online (see the Wikipedia article on RSA). Keep in mind, however, that the Wikipedia page uses different variable names than what the lecture and slide set used. You will also want to reference the Java SDK documentation, specifically the `java.math.BigInteger` and `java.security.MessageDigest` classes.

3.2 Starter Code

To simplify the assignment, and to allow easy interoperability between your code and our test cases, I have provided a significant amount of starter code in the `RSA.java` file. That code is split into three classes, the first two of which are just to hold data and have no methods of their own. Javadoc documentation is available for each of these classes (to view the Javadocs, unzip the folder and open the index.html file).

- **`CipherText`**: a class to hold the cipher text for any RSA encrypted data
- **`RSAKey`**: a class to hold a public or private (or both) key

- **RSA**: the main class that implements RSA. There are eight methods within that need to be completed for this assignment: `convertToASCII()`, `convertFromASCII()`, `generateKeys()`, `encrypt()`, `decrypt()`, `crack()`, `sign()`, `checkSign()`.

3.3 Programming Tasks

For this assignment, you must implement eight methods in the RSA algorithm. They are the ones that state "requires completion" in the comments. This assignment is to be done in Java, as you will need to use the **`BigInteger`** and **`MessageDigest`** classes.

You must start with the `RSA.java` starter. This code, described more fully below, provides various methods to ensure that the format of the key files and the encrypted files are consistent, so that I may easily test your code. Furthermore, I am going to *EXPLICITLY* call individual methods from your code, so if they are not named exactly as they are in the starter code, your code will fail those tests.

WARNING!!! The `convertToASCII()` and `convertFromASCII()` methods will be used to test your code. If they don't work properly, then NONE of your other methods will work. My unit tests, which I will use to evaluate your code, will call these to convert back and forth. So if they aren't working properly, then you will likely end up losing most of the points on the assignment. Thus, please implement them first, and test them to make sure they work properly. And you should use them in your `encrypt()` and `decrypt()` methods, of course.

You may add any other methods or fields that you would like to add. The following is required for my testing harness to work properly on your code:

1. The `main()` method exactly as provided, as I want to make sure the command line parameters are interpreted the way I expect them to be interpreted (more on this below).

2. The eight methods that you are to implement should not have their signatures changed (return type, visibility, static-ness, name, or parameter types). All these methods have throws Exception in case your implementation decides to throw an exception.

3. The two utility classes (`RSAKey` and `CipherText`) should not be modified.

4. The utility methods, which are the six methods that you don't have to implement, should not be modified. Those are: `writeKeyToFile()`, `readKeyFromFile()`, `readCipherTextFromFile()`, `writeCipherTextToFile` ↪ `()`, `getFileContents()`, and `convertHash()`.

5. As mentioned above, you may add any other methods or fields that you would like to add.

These requirements are meant to allow for easy for interoperability and to ensure that your code works with my test cases.

You must implement eight methods within the `RSA.java` file. Those methods are: `convertToASCII()`, `convertFromASCII` ↪ `()`, `generateKeys()`, `encrypt()`, `decrypt()`, `crack()`, `sign()`, `checkSign()`. Details for how they should work can be found in the the RSA lectures as well as online.

The documentation contained in the comments in the starter code details what the methods and classes do (or should do). That commenting was run through Javadoc, and the results are: CipherText, RSAKey, and RSA. ***You will want to look at this!***

3.4 Command Line Parameters

The `main()` method should not need to be modified for the final submission (feel free to modify it any way you want to test your code). It will call the appropriate methods as indicated by the command line parameters, which are described here. In almost all cases, output (progress, status messages, etc.) should ONLY be printed to the standard output if the `-verbose` option is set and should be enough that I can understand what is happening. The only time output should be printed to the terminal is when: (1) a signature does not match; and (2) an error condition is encountered (which, in theory, should not happen with my tests on properly implemented code).

Note that the command-line parameters are parsed in order—this means that if you call `java RSA -keygen 10 -verbose`, you will not get any verbosity, as that parameter was specified after the `-keygen` parameter was given. I provide a `main()` method in the `RSA.java` starter code, which can handle the parameters and again, should not be modified.

Note that normal operation (i.e., without the `-verbose` flag) is for it to print nothing—the only exception is the `-checksign` flag.

Again, the command line parameters are listed here.

You may assume (and, in fact, should) that you will only receive one of `-encrypt`, `decrypt`, `-sign`, `-checksign`, `-crack`, or `-keygen` at a time; this specifies what the program is going to do. The program should not output any messages on normal execution (it should output error messages, as appropriate, but I won't be giving any invalid combination of parameters) other than on `-checksign`. With the `-showpandq` option, it will of course output p and q. And with the `-verbose` option, it can output a lot of informational messages.

Furthermore, you may assume that I will not give you invalid input, either in the files I provide, or for the command-line parameters.

3.5 Sample Test Script

A sample usage of the program, in which Alice and Bob are sending messages, is available. This code can be found in a shell script named `test-rsa.sh`. If you are on a UNIX, Linux, or Mac environment, open a terminal, navigateto the directory where you saved the test script, and run `chmod 755 test-rsa.sh`. You can then run it via `./test-rsa.sh`. If you are on a Windows system, then you may have to copy-and-paste those commands into a command terminal (or you can probably run it in a program like PowerShell, which mimics a LInux/Unix environment). *Note that this is not a complete test suite!* Just a quick check to see if the basics work. But if your program does not work with this, then it's incomplete, and will receive a low grade.

This file creates two files (`message1.txt` and `message2.txt`), and will overwrite those files if they already exist; those two files are deleted by the last line in the script. You are welcome (and encouraged) to use other, longer, message test files–might I suggest some great speeches? However, make SURE that the text is all ASCII, since your program will likely not be able to handle UTF-8 characters (another character format)—and when you cut-and-paste, characters like the dash and quotes often do not cut-and-paste into their ASCII equivalents. Run `file message1.txt` to check if it is ASCII or not (not ASCII is typically reported as 'UTF-8').

Assuming you don't have any extraneous output (which you shouldn't), the only command that should output anything is step 12. Again, you can see the test script at `test-rsa.sh`.

Note that the last line deletes the intermediate files; during development, you may want to comment that line out to see those files.

The output should be just:

```
Signatures do not match!
```

While I am not going to try to break your program with strange combinations of command line parameters (trying to decrypt but not specifying a key), I would encourage you to put some error-checking code in your methods for your own sanity while developing the program.

3.6 Helpful Tips and Notes

3.6.1 Determining SHA-256 Hashes

We will be learning more details about hashing and encryption next week when we will look at the MD5 hashing algorithm in detail. For this assignment, you will use the SHA-256 to sign a message with RSA. You will not have to implement any of the hashing code, but your `sign()` and `checkSign()` methods will use the functions provided in the starter code.

If you want to see if a string has the same SHA-256 as a file, make sure they are *EXACTLY* the same. If the file has a ending newline (`\n`) character, and the string does not, then the SHA-256 sums will not match! You can find the SHA-256 hash of a file via the `sha256sum` command on UNIX/LInux/Mac or the `CertUtil -hashfile message1.txt SHA256` on Windows:

```
$ sha256sum message1.txt
bdf712419bb34b8c0f0d08126f191ecab5da9c0dbb4d2711d3c8eed6f5d42f2a message1.txt
```

If your system does not have `sha256sum`, you can switch over to MD5, *but only for development!* To do that, change the value of the `hashAlgorithm` variable to "MD5". You can then use the `md5` or `md5sum` commands similarly to what is shown above for `sha256sum`. ***Be sure to change it back to "SHA-256" before submission!***

Note that, given an input file, your SHA-256 hash computation should print out the exact same result as the `sha256sum` command on your favorite UNIX/Linux system. But see the warning, above, about making sure the contents are the exact same.

3.6.2 ASCII

All files (messages, keys, ciphertext, what-not) will have only printable ASCII characters, so you need not worry about binary files. But there may be whitespace as well: newlines, tabs, linefeeds, etc. *Make sure that your code does not have UTF-8 characters in it!* Given a file, you can tell what type of characters it has via the `file foo.txt` command.

3.6.3 Using a Block for RSA

- **Creating the block:**
  You will have a series of characters to put into a block for encryption (or, for decryption, to extract out of the block). There are a few ways you can do this. You can convert it to a byte array (the `String getBytes()` method does this)—each value in that array is a single (ASCII-encoded) value. You can optionally multiply the current running block by 256, then add the current value to be appended to the block. Obviously, your number will need to be in a `BigInteger`.

- **Block size determination:**
  To figure out your block size (which we'll call b)—which is the number of characters you can encode in one block—let x be the number of bits in n (found via the `BigInteger bitLength()` method). Divide $x - 1$ by 8 (the minus one is important here to prevent rounding issues). As mentioned below, you can assume that we will always use keys that support a block size of at least 2. This is for encryption only. For decryption, the block size will be indicated in the ciphertext file which, when read in by the provided `readCipherTextFromFile()` method, will be in the `blockLength` field of the returned `CipherText` object.

- **Block size minimum:**
  Each block will be a whole number of 8-bit characters, so I will not be splitting characters between blocks. Thus, if your keys can hold up to 31 bits per block (if $2^3 1 < n < 2^3 2$), I will only encode 24 bits (3 characters) in that block, not 31 bits. That being said, you can assume that all blocks I will use will allow for at least 2 characters per block. Note that the number passed in via `-keygen` is the bit size for p and q; n is roughly twice that size.

4 Submission

Submit your ***RSA.java*** file, which contains all your Java code. The compilation command on the command line will be `javac *.java`, so your RSA.java should have only one public class called RSA, and not be in a package. I will expect to see an `RSA.class` file after compilation, and will not be able to run and evaluate your code if it is not present.