Assignment 2: Public Key Cryptography

This project is due on **Thursday, February 22** at **11:59 p.m.**. Late submissions will be penalized by 10% per day. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your project early.

This is an individual project.

The code and other answers you submit must be entirely your own work. Undergraduate students are bound by the Honor System while graduate students are bound by the Graduate School's expectation of research integrity. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions must be submitted electronically via CS Dropbox.

Introduction

In this assignment, you'll add functionality to the code you wrote for assignment 1, toward the goal of implementing a secure facility for client-server communication across the Internet.

As before, we will give you some of the code you need, and we'll ask you to provide certain functions missing from the code we provide. You can download the code we are providing at: https://goo.gl/cYGQYE. Create a fresh directory and unzip the downloaded code into it. Then copy into that same directory all of the .java files from your solution to assignment 1. As before, you must not use any crypto libraries; the only primitives you may use are the ones we gave you, and ones you implemented from scratch yourself.

In this assignment you will implement three facilities, by modifying three Java code files. You will modify *RSAKeyPair.java* to generate an RSA key-pair. You will modify *RSAKey.java* to implement secure RSA encryption and decryption, and to create and verify digital signatures. You will modify *KeyExchange.java* to implement a secure key exchange. As in the previous assignment, we have provided you with code files in which some parts are "stubbed out". You will replace the stubbed out pieces with code that actually works and provides the required security guarantee. We have put a comment saying *"IMPLEMENT THIS"* everywhere that you have to supply code.

Although your solution may call on code that you wrote for assignment 1, your solution to this homework should not rely on any specific properties of your assignment 1 code. We will test your solution with our own implementation of the assignment 1 functionality. Your solution must work correctly when we do this — this shouldn't be a problem as long as you respect the API boundaries between the different classes we have given you.

Objectives

- Understand how public key cryptography works

RSAKeyPair

Your *RSAKeyPair* class should implement the following API:

```
public class RSAKeyPair {
    public RSAKeyPair(PRGen rand, int numBits)
    public RSAKey getPublicKey()              //already implemented
    public RSAKey getPrivateKey()             //already implemented
    public BigInteger[] getPrimes()
}
```

For *RSAKeyPair*, the bulk of the interesting work is performed by the constructor. This constructor should create an RSA key pair using the algorithm discussed in class. The constructor will use the PRGen called *rand* to get pseudo-random bits. *numBits* is the size in bits of each of the primes that will be used. The key pair should be stored as a pair of *RSAKey* objects.

getPrimes() is a method we've added in order to help us with the grading process. *getPrimes()* should return the two primes that were used in key generation. Typically, you would not explicitly have a method to return these primes. The primes may be returned in either order.

RSAKey

Your *RSAKey* class should implement the following API:

```
public class RSAKey {
    public RSAKey(BigInteger theExponent, BigInteger theModulus)
    public BigInteger getExponent()
    public BigInteger getModulus()
    public byte[] encrypt(byte[] plaintext, PRGen prgen)
    public byte[] decrypt(byte[] ciphertext)
    public byte[] sign(byte[] message, PRGen prgen)
    public boolean verifySignature(byte[] message, byte[] signature)
    public int maxPlaintextLength()
    public byte[] encodeOaep(byte[] input, PRGen prgen)
    public byte[] decodeOaep(byte[] input)
    public byte[] addPadding(byte[] input)
    public byte[] removePadding(byte[] input)
```

The *RSAKey* class implements core RSA functions, namely encrypting/decryption as well as signing/verification. Note that the *RSAKey* class is used for both public and private keys, even though some key/method combinations are unlikely to be used in practice. For example, it is unlikely that the *sign()* method of a public *RSAKey* would ever be used.

The *encrypt()* method should encrypt the plain text using optimal asymmetric encryption padding (OAEP) as discussed in class. It is not enough to simply exponentiate and mod the plain text. *encrypt()*, *sign()*, and *encodeOaep()* take a PRGen parameter, in case the implementation wants to use some pseudo-random bits. The *decrypt()* method should be able to decrypt the cipher text.

Your code for OAEP encoding and decoding should be in the provided *encodeOaep()* and *decodeOaep()* methods. Your other methods should call these utility methods to encode/decode when necessary. When *decodeOaep()* fails integrity checks, it should reveal this by returning null or throwing an exception. For full credit, don't forget to pad the input to the OAEP algorithm if it is too short – this is necessary to guarantee security (otherwise the exponentiated message might be smaller than the modulus).

The *sign()* method should generate a signature (array of bytes) that can be verified by the *verifySignature()* method of the other *RSAKey* in the private/public *RSAKey* pair. You should not include the entire message as part of the signature; assume that the verifier will already have access to this message. This assumption of access is reflected in the API for *verifySignature()*, which accepts the message as one of its arguments.

The *verifySignature()* method should be used by a public *RSAKey* object to verify a signature generated by the corresponding private RSAKey's *sign()* method.

The *maxPlaintextLength()* method should return the largest N such that any plain text of size N bytes can be encrypted with this key and padding scheme. Your code must correctly operate on plain texts that are any size less than or equal to the size returned by *maxPlaintextLength()*.

The *addPadding()* and *removePadding()* methods are used to pad the input to the OAEP algorithm if it is too short. **You should not call these methods from within *encodeOAEP()/decodeOAEP()*.** See below for more information on this.

KeyExchange

Your *KeyExchange* class should implement the following API:

```
public class KeyExchange {
    public static final int OUTPUT_SIZE_BYTES
    public static final int OUTPUT_SIZE_BITS
    public KeyExchange(PRGen rand, boolean iAmServer)
    public byte[] prepareOutMessage()
    public byte[] processInMessage(byte[] inMessage)
}
```

The constructor should prepare to do a key exchange. rand is a secure pseudo-random generator that can be used by the implementation. *iAmServer* is true if and only if we are playing the server role in this exchange. Each exchange has two participants; one of them plays the client role and the other plays the server role.

Once the *KeyExchange* object is created, two things have to happen for the key exchange process to be complete:

- Call *prepareOutMessage* on this object, and send the result to the other participant.
- Receive the result of the other participant's *prepareOutMessage*, and pass it in as the argument to a call on this object's *processInMessage*.

These two things can happen in either order, or even concurrently (e.g., in different threads). This code must work correctly regardless of the order.

The call to *processInMessage* should behave as follows:

- If passed a null value, then throw a *NullPointerException*.
- Otherwise, if passed a value that could not possibly have been generated by *prepareOutMessage*, then return null.
- Otherwise, return a "digest" (hash) value with length *OUTPUT_SIZE_BYTES* and the property described below.

Your *KeyExchange* class must provide the following security guarantee: If the two participants end up with the same non-null digest value, then this digest value is not known to anyone else. This must be true even if third parties can observe and modify the messages sent between the participants.

This code is **NOT** required to check whether the two participants end up with the same digest value; the code calling this must verify that property.

Getting Started

Tips This list may grow in response to Piazza questions.

- Start with *RSAKeyPair*. While it is true that it contains instances of *RSAKey*, *RSAKeyPair* does not use any of the methods that you'll be implementing in *RSAKey*.
- As in assignment 1, the spec is deliberately vague regarding how you should accomplish each task. There is a significant design component to each problem.
- Make sure to run your code with the java *-ea* flag, so that assertions are enabled.
- Use *BigInteger*:
  - Since you'll be doing math with very large integers, you'll probably want to use the *java.math.BigInteger* library class for any such operations. This class provides myriad functions that you may find useful for this assignment, particularly as *BigInteger* was originally designed with RSA implementation in mind. (Using *BigInteger* doesn't violate our rule against using external crypto primitives, because *BigInteger* provides basic mathematical functions, and not crypto.)
  - If you find yourself writing complex functions involving *BigIntegers* (e.g. manually testing primality, manually generating primes, manually finding the greatest common denominator of two numbers, manually finding d given p, q, and e, etc.), you're doing way more work than you need to. Find the appropriate *BigInteger* method.
  - One particularly useful *BigInteger* function is *modPow()*.
  - Converting back and forth between *BigIntegers* and *byte[]* arrays is a major hassle. It's surprisingly hard to get this code right. We have given you code, in the *HW2Util.java* file, that can do this.
  - For maximum elegance in *RSAKey*, your message should only be in *BigInteger* format for the purposes of exponentiating and modulusing. In other words, when you're applying OAEP, padding, unOAEP, etc., it's much easier to deal with your input in terms of *byte[]*.
- In class, we said that given public and private keys *(d, N)* and *(e, N)*, we have that $x = (x^{(de)} \ mod \ N)$, if $0 < x < N$. Thus, if you're going to use the built in *BigInteger* functions to encrypt and decrypt, it is important that you represent your input message as a positive *BigInteger*.
- An *RSAKey* object does not know if it is "private" or "public". Indeed, it is even possible to sign messages using a public key or encrypt using a private key, though neither of these strange operations are likely to be useful in practice.
- We recommend that first you get your code working for the case where all inputs are full sized, then modify your code so that it handles padding. When you implement padding, the relevant code should be in the provided *addPadding()* and *removePadding()* methods. Your other methods should call these utility methods to pad/unpad when necessary.

- There is a bit more programming this week. Our reference solutions are 45, 151, and 84 lines of code (including everything, even comments, whitespace, brackets, etc.) for *RSAKeyPair*, *RSAKey*, and *KeyExchange* respectively.
- If your think that your solution to assignment 1 is flawed, you will want to correct these mistakes before testing your implementation of assignment 2. Alternatively, you can use a pre-compiled version of our sample solution for assignment 1, which you can download at: https://goo.gl/3dpYwL.

Submission Checklist

Upload to CS Dropbox the files listed below. Make sure you have the proper filenames and behaviors.

RSAKeyPair

`RSAKeyPair.java`	A file containing your implementation of the *RSAKeyPair* class.

RSAKey

`RSAKey.java`	A file containing your implementation of the *RSAKey* class.

KeyExchange

`KeyExchange.java`	A file containing your implementation of the *KeyExchange* class.