Assignment 3: Authenticated Key Exchange

This project is due on **Thursday, March 1** 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 Assignments 1 and 2, to reach 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 here: https://goo.gl/XevDyX. Create a fresh directory and unzip the downloaded code into it. Then copy into that same directory all of the *.java* files from your solutions to Assignments 1 and 2. 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. If you'd like to use our reference solutions for assignment 1 and 2, you may download them after the submission deadline of assignment 2 here: https://goo.gl/aUgZ17.

In this assignment you will implement a secure channel abstraction that can be used by two programs, a client and a server, to communicate across the network, with the confidentiality and integrity of messages guaranteed. We have given you a class *InsecureChannel* which implements a channel that works but is not secure: everything is sent in unprotected cleartext. We have also given you stubbed-out code for a class *SecureChannel* that extends *InsecureChannel* and (once you have modified it) will protect security and confidentiality.

To facilitate the testing process, we have provided a few test files. The first is *Util432s.java*, which provides a few methods used by the other testing files (feel free to use these methods for debugging). The second is *ChannelTest.java*, which provides a demonstration that the *InsecureChannel* class works correctly. The third is *SecureChannelTest*, which you can use to test your *SecureChannel* class (once implemented). Note that this class does NOT test security properties, instead testing only basic functionality. Note the commented out lines which give an example of how you can *Util432s* for debugging. The fourth is *InsecureChannelDebug*, which is a special version of *InsecureChannel* which provides the ability to echo channel transmissions to the screen. To use this class, simply rename the file *InsecureChannel.java* and place it in the same directory as *SecureChannel.java*.

AuthEncryptor

Your *AuthEncryptor* 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()
}
```

This class is used to perform authenticated encryption on values. Authenticated encryption protects the confidentiality of a value, so that the only way to recover the initial value is to decrypt the value using the same key and nonce that was used to encrypt it. At the same time, authenticated encryption protects the integrity of a value, so that a party decrypting the value using the same key and nonce (that were used to encrypt it) can verify that nobody has tampered with the value since it was encrypted.

Code that uses *AuthEncryptor* will be required to pass in a different nonce for every call to encrypt. The *AuthEncryptor* class is not required to detect violations of this rule; it is the responsibility of the code that uses *AuthEncryptor* to avoid re-using a nonce with the same *AuthEncryptor* instance.

If *includeNonce* is true, then the nonce should be included (in plaintext form) in the output of encrypt. If *includeNonce* is false, then the nonce should still be used in calculating the output, but the nonce itself should not be copied into the output. (Presumably the party who will decrypt the message already knows what the nonce will be.)

AuthDecryptor

Your *AuthDecryptor* class should implement the following API:

```
public class AuthDecryptor {
    public AuthDecryptor(byte[] key)
    public byte[] decrypt(byte[] in)
    public byte[] decrypt(byte[] in, byte[] nonce)
}
```

The value passed as *in* will normally have been created by calling *encrypt()* with the same nonce in an *AuthEncryptor* that was initialized with the same key as this *AuthDecryptor*.

If the integrity of the input value cannot be verified (that is, if the input value could not have been created by calling *encrypt()* with the same nonce in an *AuthEncryptor* that was initialized with the same key as this *AuthDecryptor*), then this method returns null. Otherwise it returns a newly allocated byte-array containing the plaintext value that was originally passed to *encrypt()*.

If the nonce is included in the message, then the message should be decrypted with *decrypt(byte[] in)*. Otherwise, the nonce should be provided along with the ciphertext to *decrypt(byte[] in, byte[] nonce)*.

SecureChannel

Your *SecureChannel* class should implement the following API:

```
public class SecureChannel extends InsecureChannel {
    public SecureChannel(InputStream inStr, OutputStream outStr,
       PRGen rand, boolean iAmServer,
            RSAKey serverKey) throws IOException
    public void sendMessage(byte[] message) throws IOException
    public byte[] receiveMessage() throws IOException
}
```

The constructor will contain the vast majority of your code. Its role is to set up the secure channel such that the *sendMessage* and *receiveMessage* methods can do their jobs. These methods should provide authenticated encryption for the messages that pass over the channel, ensuring that messages arrive at the receiving end in the same order that they were send on the sending end. Furthermore, when the client is setting up its channel, it should also authenticate the server's identity, and should take whatever steps are necessary to detect any man-in-the-middle. If one of the two parties (server or client) detects a potential security problem during channel construction, that party should close the channel by calling *close()*. You can assume the *serverKey* (public key) passed to the constructor of *SecureChannel* on the client side of the communication is verified externally in some way (for example via a trusted certificate).

The underlying *InsecureChannel* will normally deliver messages in the same order they were sent. But note that an adversary might try to reorder messages. *receiveMessage* should return null if an invalid or out-of-order message shows up.

Getting Started

Tips This list may grow in response to Piazza questions.

- Start by looking at *ChannelTest.java*, which will provide you with a better understanding of how *InsecureChannel* (and *SecureChannel*) are intended to be used. In particular, two instances of *InsecureChannel* will be created (one for the server->client channel and one for the client->server channel), each of which connects up two data streams (one input and one output data stream). Messages are sent through the channel using the *sendMessage()* method, and whenever a message is sent via a channel, it stays there until a corresponding *receiveMessage()* call is made. Luckily for you, you won't need to think about *InputStreams* or *OutputStreams* at all. That is all taken of in *InsecureChannel.java* and in the main function of the *ChannelTests*.
- If you're unfamiliar with *InputStreams* and *OutputStreams*, don't worry, you won't be dealing with them very closely. Same goes for runnable classes and threads.
- Another thing you might try is copying *InsecureChannelDebug* over *InsecureChannel* and running *ChannelTest*, which will let you see the raw traffic being sent over the channel.
- From there, look at *SecureChannelTest.java* to get a feeling for how the *SecureChannel* instantiations differ.
- From here, you should design a threat model. Carefully consider everything that your adversary is trying to do that you'd like to prevent. Place this threat model and mitigations in your README file.
- Once you feel comfortable with your threat model, consider what tools you have available from assignments 1 and 2. Your design should be modular, where specific classes solve specific subtasks. Sketch everything out schematically and try to find ways to compromise the security of your design. Perhaps the best way to do this is to start your design early, sleep on it, and then come back later to analyze it from a perspective that has been freshened by sleep.
- Once you feel comfortable with your overall design, implement and test.
- For reference, our solution for *SecureChannel* is a mere 87 lines of code, *AuthEncryptor* is 66, and *AuthDecryptor* is 78. Unlike assignments 1 and 2, you should not run into any programming challenges while writing *SecureChannel* (though writing additional tests beyond *ChannelTest* might be tough). Your code should be very straightforward and easy to code up from your design.
- In *SecureChannelTest*, if a thread calls *readMessage* and there is no message available, it will wait until a message becomes available.

Submission Checklist

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

README

`README`	A file containing your description of your implementation.

AuthEncryptor

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

AuthDecryptor

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

SecureChannel

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