Project #2: Secure Email System
Due: Tues, November 29th in class

(CAETE students may email their project to Martin)

As advertised, in this project you will provide a secure email system for use within this class. Your system will allow you to send and receive encrypted and signed email with any other student in the class (and with your instructor and grader).

We are going to use both symmetric-key and public-key techniques in this project, thus tying together several of the concepts discussed in lecture. As usual, we'll use OpenSSL as our toolkit, either via the command-line interface (easiest) or via system calls (you'll need the OpenSSL book for this!) The program you write will have three main functions:

- A mini-database utility to keep track of certs you have acquired from the web site http://ucsu.colorado.edu/~cochranm/certs.html
- A method to send encrypted and signed email
- A method to verify and decrypt received email

Message Format

The message format is as follows:

-----BEGIN CSCI 6268 MESSAGE-----
<session pwd encrypted under target's public key>
<blank line>
<message encrypted under session pwd above>
<blank line>
<signature of above content with newlines omitted>
-----END CSCI 6268 MESSAGE-----

Notes:

1. First -----BEGIN CSCI 6268 MESSAGE----- must appear exactly as shown; this is the indicator that the message begins immediately after this line. (This allows the message to be embedded in a bunch of other text without confusing the recipient's parser.)

2. The next line is the session password encrypted under the target's public key. This password is a random string of 32 characters using A-Z, a-z, and 0-9 generated by the sender; the sender then encrypts his message with AES in CBC mode using this password.

3. There is a blank line, followed by the AES-CBC encrypted message in base64 format. This is followed by another blank line.

4. Next comes the signature of the sender which is generated using the sender's private key. This signature will be the RSA signature of the SHA-1 hash of every line above from the first line after the BEGIN marker to the line just before the blank line ending the message. Do not include newlines, as these are different between DOS and Unix (in DOS we have <cr><lf>, whereas in Unix it's just <lf>.)

5. Finally, -----END CSCI 6268 MESSAGE----- concludes the encrypted message.

The Certificate Database

Your program should maintain a simple catalog of certs which you have collected from the web site. You may store them in whatever format you prefer (a flat file is the simplest, but if you prefer to use MySQL or something fancier, be my guest).

Notes:

1. A cert should always be verified using the CA's public key before being inserted into the database.

2. A cert should always be verified using the CA's public key after being extracted from the database (to ensure someone hasn't tampered with it while you weren't watching).

3. You need not store the person's email address in your database since this is embedded in the cert, but it might be easier to go ahead and store the email addresses as an index field in the file. Of course, you must not rely on these index names as the validated email addresses; always make sure the email in the cert matches!

4. There is some difficulty with the re-writing of email addresses; you can be liberal in how you handle this. For example, email from me may appear to you as John.Black@colorado.edu; this is the same person as jrblack@cs.colorado.edu listed in my cert, and you can feel free to make this obvious mapping in your program.

Sending Secure Email

Your program should accept a plain-text message along with a destination email address and output an encrypted and signed message as we described a moment ago. Here is the algorithm:

1. Get the cert of the target from the database, using the email address as the index; if the email is not there, you must extract it from the web page.

2. Verify the signature on this cert for your email target.
3. Generate a 32-character passphrase. Normally we would use a very strong random-number generator for this, but feel free to use random() or the rand function of OpenSSL if you like.

4. Encrypt the message with AES in CBC mode with the session key and a random IV (OpenSSL does this for you). Use base64 encoding, and save the output.

5. Encrypt the session password with the target's public key.

6. Sign the stuff generated so far as described previously, using SHA-1 and your private key (you will need to type in your passphrase to do this). Remember to exclude newlines.

7. Format and send.

Receiving Secure Email

This is how you will process incoming secure email:

1. Obtain sender's email address from mail header.

2. Find sender's cert in your database, or obtain from the class website. Verify sender's cert is signed by CA; output sender name from the cert (not from the email header!)

3. Verify signature on received message using SHA-1 and public key of sender. If invalid, reject the message. Else, continue.

4. Decrypt session key with your private key (you will need to type in your passphrase for this).

5. Use session key to decrypt message; print out resulting message.

Using OpenSSL

As I mentioned, we need to use the same SSL commands in order to maintain interoperability. Therefore, I'm listing some very useful ones below. This does NOT give you everything you'll need for the project, so there is still some work you'll need to do in order to get the job done. Sifting through the OpenSSL website will do the trick. ☺

Ok, suppose our password (32-char random passphrase) is in the file "key."

% cat key
ABCDEFGHabcdefgh08090a0b0c0d0e0f

To RSA-encrypt "key" with a public key from test-cert.pem:

% openssl rsautl -in key -inkey test-cert.pem -encrypt -certin

But there is a problem: this comes out all garbled. So figure out how to base64-encode this output, then write it to a file and you have the RSA-encrypted session key (passphrase).

Now, using the SSL docs, figure out how to get the key back again.

You already know how to AES-CBC-encrypt with a passphrase (don't put it on the command line!), so we'll skip that part. Finally, you'll need to sign everything but the newlines in the two chunks (RSA-encrypted passphrase and AES-CBC-encrypted message). As usual, we'll do this with hash-and-sign. First use SHA1 to hash:

% openssl sha1 test-msg.enc

This outputs a digest, which (for most versions of OpenSSL) includes the file name like this:

SHA1(test-msg.enc)= 7784269c9ae197b3495853c8fdb3b8633db27ab0

But do not include the part before the = sign, because we don't want the filename to matter (it's a temporary filename!). Just take the digest itself and sign it. In other words, only sign the

7784269c9ae197b3495853c8fdb3b8633db27ab0

But how to sign? Use the rsautl sign function; here's how I would sign with my private key (this signs input from stdin):

% openssl rsautl -sign -inkey john-priv.pem

Once again, the output needs to be base64 encoded, so do that again. Then append that to the above two chunks, wrap up all three chunks, and send.

You will have to figure out how to verify signatures of the above form (another rsautl command… refer to the docs).

What to hand in

You will be sent two test messages from me. One has a good signature, the other does not.

Hand in three things:

1) Source for your program
2) The output of your program when running on the two test messages I sent you
3) The output of your program when composing a reply (of your choice) to me

For step (3) do **not** actually send me the email. Just show how your program constructs the message, and show the formatted message that **would** be sent to me if you sent it. (This may require that you disable the email part of your program just for a moment.)

As usual, hand this in as hardcopy in class on the due date, Nov 28th. If you are a distance student, you may email your homework directly to Martin.