# CS 165 SSL Project - Fall 2012

## 1 Overview

The goal of this project is for you to become familiar with programming many of the operations used in secure communications schemes such as SSL.

For this project, you will implement a client/server application in C/C++ using the OpenSSL library. In this scenario, the client has the server's "certificate" (just its RSA public key here), and wants to retrieve a sensitive document from the server. The client first establishes an SSL connection to the server, then verifies the server's authenticity by sending the server an encrypted (RSA public) random challenge, which the server decrypts (RSA private), hashes (SHA1), signs (RSA private), and returns.

The client then sends a file request to the server, specifying the filename of a document to be retrieved. The server returns the document over the SSL connection, and the client displays it.

## 2 Implementation Details

Your application will perform the following basic OpenSSL operations when establishing a connection: initializing, creating a context, creating an SSL object, creating a BIO object from a socket, and connecting the BIO and SSL objects. You may review instructions/examples from Ch. 5 of the OpenSSL online book, or the *Introduction to OpenSSL Programming* PDFs posted. The resources listed in Lab Assignment 7 may also come in handy.

Use the PEM files you submitted as part of Lab 7 as the server's RSA keys. The client has access to only the public key, while the server has both. The text file you submitted for Lab 7 must be one of the files available for request by the client. You may make other files available also, but only this one is required.

You will first establish an SSL connection between the client and server using the Anonymous Diffie-Hellman (ADH) key-sharing scheme. All your network traffic will flow through this connection. OpenSSL handles most of the mathematical details of ADH and the underlying network communication of the SSL connection for you. Once the connection is established, you will perform authentication using random challenges and the server's public key.

To keep the assignment simple, you are not required to use the certificate in the key exhange itself. Thus the key exchange is still vulnerable to a man-in-the-middle attack. Since your authentication is ad-hoc (i.e. not part of the key exchange), it tells you only that the server is at the other end, but there may still be an interceptor. To allow OpenSSL to use ADH, despite its security issues, you must use the SSL_VERIFY_NONE flag when setting up SSL_CTX_set_verify in the SSL context object.

### 2.1 Client-Side Steps

The client should execute as follows:

1. Establish an SSL connection with the server.
2. Seed a cryptographically secure PRNG and use it to generate a random number (challenge).
3. Encrypt the challenge using the server's RSA public key, and send the encrypted challenge to the server.
4. Hash the un-encrypted challenge using SHA1.
5. Receive the signed hash of the random challenge from the server, and recover the hash using the RSA public key.
6. Compare the generated and recovered hashes above, to verify that the server received and decrypted the challenge properly.
7. Send the server a *filename* (file request).
8. Receive and display the contents of the file requested.
9. Close the connection.

The client application should be executed as follows:

```
your_client_app_name -server serveraddress -port portnumber filename
```

### 2.2 Server-Side Steps

The server should execute as follows:

1. Wait for client connection request, and establish an SSL connection with the client.
2. Receive an encrypted challenge from the client and decrypt it using the RSA private key.
3. Hash the challenge using SHA1.
4. Sign the hash.[1]
5. Send the signed hash to the client.
6. Receive a filename request from the client.
7. Send the (entire) requested file back to the client.
8. Close the connection.

The server application should be executed as follows:

```
your_server_app_name -port portnumber
```

[1] We sign a hash of the challenge instead of the challenge itself to prevent an attack. If the client sends the server a challenge that is actually a ciphertext, and the server signs it directly using his RSA private key, he has just decrypted the ciphertext and returned the plaintext to the client!

# 3 Requirements

1. You must use C/C++ language to implement your application. Your code should be clearly written and well documented. Include a README file describing how to compile and run your code.

2. Organize the files for the client and server into separate directories. The client and server applications should be able to run on separate machines.

3. You must use the PEM files and the security document submitted in Lab 7.

4. Both parts of the applications should:

   (a) Print informative messages to the console after each step in Sections 2.1 and 2.2 (e.g. "CLIENT STEP 3").
   (b) Place comments in your code to denote the steps in Sections 2.1 and 2.2.
   (c) Check for errors during communication and display appropriate, readable error messages. After printing error messages, the connection should be terminated. See the end of Lab 7 for OpenSSL error printing instructions. Some "expected" errors may require you to make your own, more relevant, messages.

5. You are *required* to use github for your project. We want to see that incremental progress is made on the code. Do *not* check your code in all at once. We will review your git repository, and you *will* be graded down for not using git or for not doing incremental check-ins. There is nothing wrong with checking in partial or incorrect code. Only the final version needs to work. *BUT MAKE SURE YOUR COMMIT MESSAGES ARE MEANINGFUL!*

6. Besides using github, upload your final project submission using iLearn.

7. Your submission *must* be your own, original work. You may discuss concepts and OpenSSL library nuances with other students, but sharing code will result in a failing grade. We use automated tools to check for unauthorized collaboration.