TLS Implementation Project

CS161 Staff

Due Tuesday, November 18 at 11:59pm

1 Introduction

This project will help you get very intimate with the SSL handshake. We will be implementing our own version of it, appropriately named the TLS (Terribly Lacking Security) handshake. This protocol is based off the same general idea of the real handshake, albeit with far fewer messages. It emphasizes the main concepts of the SSL handshake that you learned in class, while allowing you the joy of implementing those concepts in C code.

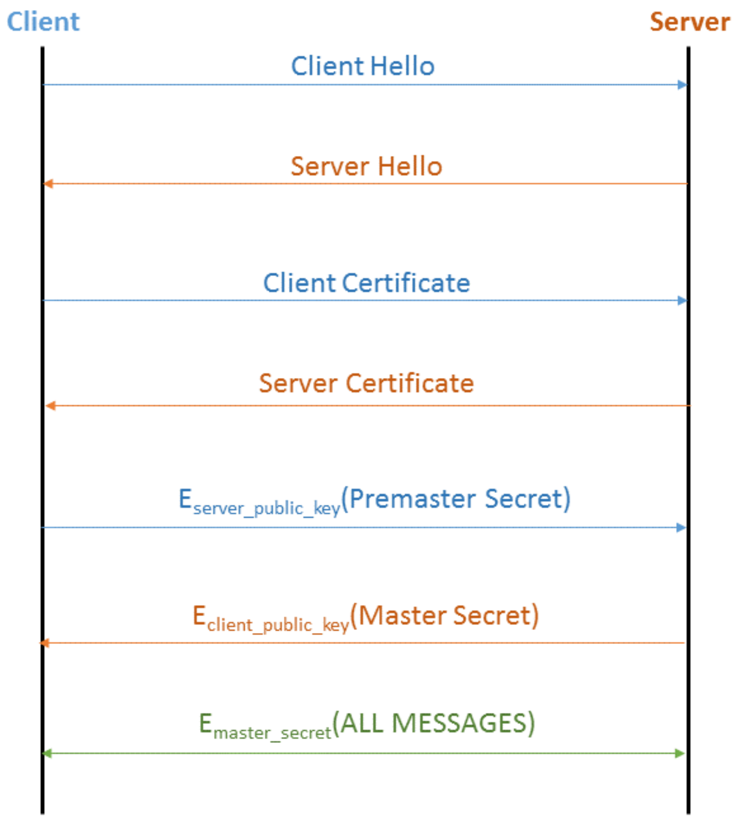
Your TAs have set up a server on torus.cs.berkeley.edu which accepts incoming connections from the client we have provided you. Once the project is finished and you can successfully establish a handshake with the torus server, you can send messages to the server by typing into the terminal.

2 Terribly Lacking Security Handshake

2.1 Overview

One minor differences between the SSL handshake we taught you in class and our Terribly Lacking Security handshake is that the client has a certificate in our handshake. As we taught you, the client generates the premaster secret and sends it over to the server (encrypted, of course!) for the server to use.

Below is a diagram entailing the message exchange order for the TLS handshake. Encryption is denoted as E; it is up to you to determine which ones are symmetric key and which ones are assymetric key encryption. Look carefully at the keys used in the encryption (the subscript to the E).



Client
Server
Client Hello
Server Hello
Client Certificate
Server Certificate
E_server_public_key(Premaster Secret)
E_client_public_key(Master Secret)
E_master_secret(ALL MESSAGES)

2.2 Messages

You will be implementing the following messages for this project:

- **Server/Client hello**: contains a type, random number (used in computing the master secret) and cipher suite.
- **Server/Client certificate**: contains the certificate signed by a Certificate Authority's private RSA key. The certificate includes the name, public key exponent, public key modulus and the expiration date.
- **Premaster Secret**: contains the randomly generated premaster secret encrypted with the server's public RSA key (obtained by reading the certificate). Both this message and the master secret message should be in the form of a hex character string with no prefix, i.e. "E0AF445C". No padding with zeros is necessary. Hint: This is how GMP returns a hex string.
- **Master Secret**: Both the client and the server compute the master secret from the premaster secret separately, but this message verifies that both sides esablished the same key. The client should check that the value from

this message, once decrypted with the client's private RSA key, matches the value computed locally. This value should be parsed and stored as binary to be used as the encryption/decryption key for sending messages with AES (or any other algorithm if it were supported in the cipher suite, but our client and server only support one cipher suite).

- **Messages**: Once the handshake is done, you can send messages encrypted using the master secret as the symmetric key.

Note that all messages also have a type, as defined in **handshake.h**. To see the structure of these messages, as well as the required constants, please refer to this header file.

3 Your Task

Finish implementing **client.c**. You will be running the client and connecting to the server, which is running on torus.cs.berkeley.edu. The port number is taken care of in the skeleton.

IMPORTANT: Before writing any code, thoroughly read and understand all of **handshake.h**. All of the structs, message formats, and important length values are defined in **handshake.h**.

3.1 Encryption

We will use the solution to Project 0. The TLS handshake requires public key encryption. In our cipher suite, we'll be using RSA. Fill in the **perform_rsa()** function in **client.c** with your solution to Project 0. If you didn't manage to finish Project 0, use the posted staff solution. We encourage using your own solution if it works, for a better sense of ownership.

3.2 Sending/Receiving Messages

You will need to implement the functions **send_tls_message()** and **receive_tls_message()**. These functions take in pointers to messages of the forms defined in **handshake.h**. For the send function, you just need to send the message on the specified socket. For the receive function, you should receive the message off of the socket and store it at the specified pointer. Note that it is possible that you may receive an error message rather than the message type that you are expecting. Your receive function should be able to handle this case. Use these functions for sending and receiving all handshake messages in the main function. You will find the system calls **read** and **write**, used for socket I/O, helpful for completing these tasks.

3.3 Certificate Decryption

The certificates in this protocol are signed with a certificate authority's private key. You may remember that in project 0, we decrypted these certificates using the CA's public key to get the plaintext. Here you should fill in the function **decrypt_cert()** to get the plaintext. This function decrypts the certificate and stores the plaintext in the specified pointer. The certificates are sent as a hex character string of the form "0x0236FA23C". Note that the certificates are prefixed with a "0x" so GMP knows that it is a hex string without specifying the base. The server will also expect your certificate to be in this form.

3.4 Generating the Secrets

The master secret is generated from a concatenated hash of the premaster secret, the client hello random number, the server hello random number, and the premaster secret again (the formula is show below). Fill in the function **compute_master_secret()** to complete this task.
The client random and premaster secret are simply randomly generated numbers. We have provided you a random number generation function called **random_int()** in **client.c** which will generate a suitable random number for the client hello as well as the premaster secret.

Master secret $= H(PS||\text{clienthello.random}||\text{serverhello.random}||PS)$

3.5 Decrypt Master Secret

Implement the function **decrypt_verify_master_secret()**. This function should use the client's private key, input through the command line, to decrypt the master secret message. It stores the value in an mpz_t type for use in comparing with the locally used value.

3.6 Handshake

Once you have completed the tasks above, you can now use these functions to implement the handshake in the **main()** function. You should follow the order of the messages as described above, alternating between sending and receiving messages. The server will not preemptively send any message (i.e. it will not send the hello and certificate before receiving your certificate in a correct run of the protocol). However, remember that you can receive an error message at any time and it is possible that an unexpected message may come in on the socket. Your program should handle errors gracefully. Check outputs of functions for unexpected behavior.

4 Running the code

To run the program, first compile using the provided **Makefile**, then use the command:

```
# ./client -i <server_ip_address> -c <client_certificate> -d <client_private_exponent>
-m <client_modulus>
```

- -i: The program takes in the IP address of the server that it is attempting to connect to. You should be able to find the IP address of the server on your own.
- -c: The certificate for the client, signed by our certificate authority.
- -d: The private RSA key exponent for the client.

- -m: The private RSA key modulus for the client.

You can find the corresponding files for the certificate and key values in the zip folder.

Once the handshake is completed, you can type messages into the command line. When you hit enter, the program will encrypt the message and send it to the server. If the handshake completed correctly and the keys are set properly, then the server will receive your message and respond with a message of its own. The message that the server sends will be printed to the screen and should appear as plaintext English.

5 Write Up

Include your answers to the following questions in a file named `tlsproj.txt`.

1. All messages in this protocol have a type field sent in plaintext. If an attacker can alter packets being sent to and from the server, explain how it can launch a DoS attack on the server or client in a way that does not reveal it's involvement.

2. Look at the function **random_int()**. How are the "random" numbers chosen? Can an attacker learn any information about our system or other random values if they know our method? Suggest a way that a man-in-the-middle might be able to use this to break our encryption. Tip: try printing random values as they are chosen during the handshake.

3. We have talked about a downgrade attack in class before. Assuming that the server and client supported multiple cipher suites (some weaker than others), show how a downgrade attack might be possible on the Terribly Lacking Security handshake. Then suggest a method or adaptation to the handshake that would mitigate a downgrade attack.

4. (Extra Credit) List as many security flaws in the Terribly Lacking Security protocol/implementation as you can find and suggest ways that you might fix them. (Note: The flaws discussed above do not count)

6 Submission

For the submission, the only C file that you need to submit is **client.c**. For the autograder, we will be using our own versions of the various header files. **It is extremely important that your code not depend on any changes to the header files. If you change these files, you risk losing points.**

To submit, run the command `submit tlsproj` in the directory containing your solution and write up.

Files to submit:

- **client.c**: Contains all of the code for the project.

- **tlsproj.txt**: Contains the answers to the writeup.