RCOMP - Redes de Computadores (Computer Networks)

2023/2024

Theoretical-practical lesson 11

- Network security.
- SSL/TLS network programming.

Network security and secure communications

Networks are exposed to all kinds of malicious attacks. This must be in focus when network application protocols are designed, and thus security must be enforced in network applications.

A network application dialogues with other remote network applications (running in remote nodes). For that, uses either UDP or TCP, and knows the remote node's address (IPv4 or IPv6) and the port number. What happens in networks between the nodes is totally out of control of network applications.

Recapping: we already knew networks are unreliable, now we must also assume they have human intelligence dedicated to damaging the transactions between network applications.

Regarding attacks, they encompass all sort of actions harmful to network applications. An attacker could simply make the network inoperative (e.g., by cutting a cable), network applications can't do much in this case.

Most attacks are more subtle, to start with, sniffing is a big issue. We must assume any information sent through a network can be read by attackers, and that will not be noticed by applications because contents are not being changed.

Authentication, privacy and integrity

To fully guarantee secure communications between two applications, there are three related facets that must be considered:

- Authentication – ensuring attackers can't impersonate a licit application.
- Privacy – ensuring attackers can't read data.
- Integrity – ensuring attackers can't change data.

Authentication is really the base stone, if not sure talking with the intended counterpart, other facets become fairly irrelevant.

Proving the identity is most often based on possessing a secret nobody else knows. For instance, a user's password, or a Pre-Shared Key (PSK) for a symmetrical cypher. In these two cases, it's required both sides know the secret, and nobody else knows it.

When using public key cryptography, the sender's identity is proven by encrypting with the its private key (creating a digital signature), the identity is then checked at the receiver by decrypting with the sender's public key (checking the digital signature).

Privacy

Given that there's no access control to data traveling through the network, the only option to ensure privacy is by encrypting.

To operate, symmetrical key cyphers require the same Pre-Shared Key (PSK) to be already available on both authentic sides, and unknown to others.

Asymmetrical key cyphers use different keys for encryption and decryption.

Integrity

The integrity facet is similar to error detection, a message authentication code (MAC) is appended to data to be sure it hasn't changed in transit.

Under the security point of view, things get far more complex because intentional malicious actions are to be expected. Thus, an attacker could change both the data, and the validation code as well, so they would be kept matching. How this issue is addressed, depends on what solutions are in place for authentication and privacy.

Integrity checking is based on **hash functions**. Hash functions receive a whatever size block of data as input and produce a fixed size set of bits output that represent the input. The output is called, the **hash code** or the **digest**, any small change in input will drastically and unpredictably change the output.

Hash functions

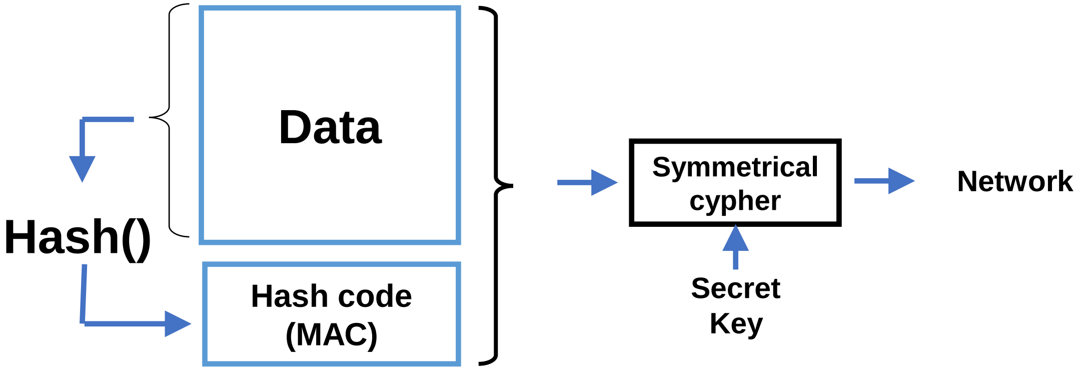
Hash functions are irreversible (one-way), they can intake any amount of data, and yet, they always produce the same number of output bits (the hash code).

Its behaviour is deterministic (the output is always the same for the same input). However, finding an input content, to match a given output hash code, it’s almost impossible.

One characteristic of a hash function, is the number of bits it returns (the hash code size). Ignoring other factors, the bigger the code, the harder will it be to attack by using brute force.

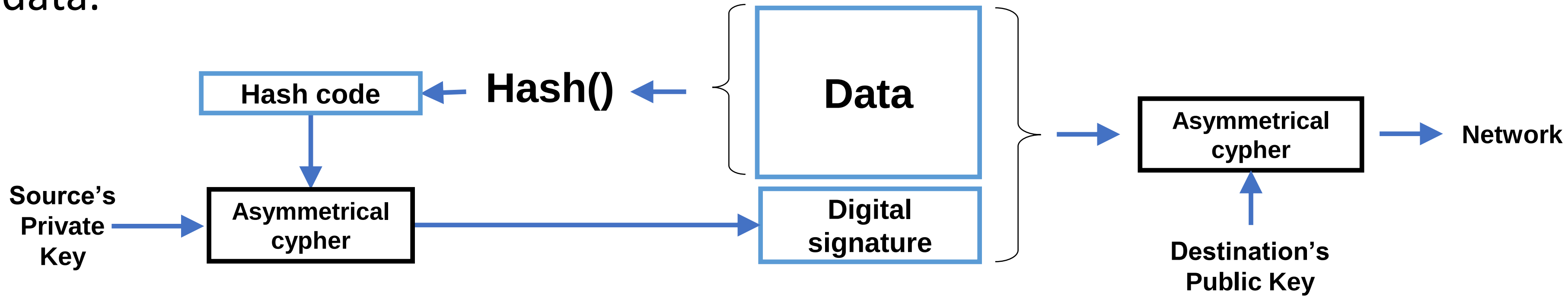
For the purpose of validating data integrity, the hash code needs to be protected against tampering.

If both the data and the hash code are being sent through a **symmetric key cypher**, then that protection is already in place.



Hash functions usage scenarios

An **asymmetrical key cypher** may also be used, but then the hash code must be encrypted using the sender's private key. And this is a **digital signature** of that data.



We can imagine an additional scenario where integrity is required, but privacy is not a requirement. This is the case, for instance, of AH in IPsec.

Then we could simply use the above scenario and omit the final, right side, encryption. If a Pre-Shared Key is available, then a symmetrical cypher could be used to encrypt the hash code only.

MAC algorithms do that directly, they are based on hash functions to produce a message authentication code (MAC), encrypted with a provided secret key. To authenticate the message, the same secret key is required.

Network applications authentication

When developing network applications, most often, privacy is vital for network exchanges. Nevertheless, authentication is a vital first step.

There are a number of approaches to authenticate applications:

1. **Symmetrical key cipher**

Applications that are to trust each other are given a same secret key. An application is authentic if is able of exchanging information encrypted with that key. This may encompass more than two applications, all sharing the same secret key. The big issue it’s how to safely provide the same secret key to all authentic applications and no others.

2**. Self-signed public key certificates**

Each application has its own self-signed certificate, in each application the counterpart´s certificate is added as trusted. If there are more than two applications, certificates of all the others must be added in each. It’s a pretty safe solution, and yet certificates have to be added manually on each node.

3. IP node addresses-based authentication

If it's known an application is running on node with a static IP address, that might be used to authenticate incoming traffic. But there're several issues. First, it's exposed to spoofing attacks, therefore, to be safe, it must be deployed in a network under tight control (e.g., within a DMZ).

Regarding the spoofing attacks (source IP counterfeiting) menace, it may be softened if double direction dialogues are enforced, this is because by using a fake source IP address the attacker doesn't receive responses to made requests. For instance, the use of TCP instead of UDP turns spoofing attacks harder due to the three-way handshake to establish the connection, encompassing random sequence numbers.

If the node bears the possibility of common users running applications, then it must also be assured the incoming traffic is not from such an application. The simplest approach is checking if traffic is coming from a reserved port number (below 1024) because common user are not allowed to use those.

To summarize, to enforce IP node addresses-based authentication, both network and the node must be under tight control of a trusted administrator.

4. Valid public key certificate

All options exposed so far, require a previous knowledge between applications and coordinated administrative settings on both sides.

However, this is not the case of a server to be made publicly available to anonymous clients. Most often, in such a scenario, the server isn't interested in authenticating client applications, perhaps it's more interested in authenticating users latter when the safe communication is already established. Nevertheless, for the client, authenticating the server is vital, otherwise it might be under a MITM (man in the middle) attack, and talking with a fake server impersonating the real one.

The server application is not aware of clients' existence before it's contacted by them, client applications know about the server's existence by knowing the server's node DNS name only.

This is solved by the use of a valid public key certificate by the server application, <u>the public key certificate associates the public key to the server DNS name</u> (CN attribute in the subject field).

If the certificate is valid (including being issued by a trusted CA) then the client has the guarantee that by encrypting data with that public key only the application running on the referred DNS name node will be able to decrypt.

Public key certificates validation

Network applications, most often client applications, need to check if a public key certificate they receive is valid, beyond validity dates included in the certificate (*not before* and *not after*), this encompasses two vital checks:

1st The issuer must be a trusted CA, this means by following the certificate chain, a certificate classified as belonging to a trusted CA is found. In other words, that certificate is locally stored in the list of trusted certificates (certificate verification storage).

2nd The CN attribute in the certificate's subject field has to match the DNS name of the node intended to be the counterpart (e.g., intended server).

Depending on the used API, these validations may be provided through suitable functions/methods.

For self-signed certificates, manually placed by the administrator in each side's certificate verification storage, the first validation is of course already established. The second validation could, in this case be omitted, if we want the counterpart authenticated application to be able to run on different nodes or in a node with a dynamic IP address.

The SSL/TLS protocol

This protocol is dedicated to establishing secure communications between two network applications. It operates by negotiating a set of mechanisms (known as cypher suite) to be used in order to ensure authentication, privacy, and integrity.

SSL/TLS was designed to transform unsafe application protocols into secure application protocols, it’s enforced immediately once the dialogue starts, prior to the original non-secure application protocol starts operating.

By using SSL/TLS insecure application protocols are turned secure, they are usually referred by the original name with an S suffix (e.g. HTTP becomes HTTPS). Yet the original insecure application protocol suffers no change whatsoever, simply data is sent and received through SSL/TLS provided functions/methods, instead of standard send/write and receive/read methods.

SSL/TLS operates over TCP or UDP, though is easier to implement over TCP than over UDP (TLS over UDP is known as DTLS - Datagram Transport Layer Security). Regarding the first vital step of authentication, either a pre-shared key or public key certificates may be used. We will be focusing on the later as that’s the only option for anonymous clients contacting a public server through its DNS node name.

The OpenSSL library (C language)

When developing network applications in C language, in Linux systems the OpenSSL library may be used to attain secure communications. This library provides a large variety of cryptographic functions, and support to the SSL/TLS protocol. Only a few will be referred here, related to establishing secure communications over TCP, based on public key certificates authentication.

Programs will have to include the relevant header files (e.g., openssl/ssl.h), and then be linked to the necessary libraries (-lssl –lcrypto).

SSL/TLS only comes to play once communications are established, thus for TCP, after the client connects to the server, and after the server accepts a client's connection. So basically, we have insecure TCP connections and SSL/TLS will transform them into secure TCP connections by a coordinated effort on both sides (client and server).

To establish the cypher suite, SSL/TLS uses the client-server model, one application is required to assume the SSL/TLS client role and send the Client Hello message, the other application must assume the SSL/TLS server role, receive the Client Hello and send back the Server Hello message. Usually, the TCP client is the SSL/TLS client, and the TCP server is the SSL/TLS server.

The OpenSSL library – context preparation

To be able to latter secure the TCP connection, a context to store SSL/TLS settings must be created in the first place, this encompasses defining a method, and thus, establishing the role. For a client by calling **TLSv1_2_client_method()**, for a server by calling **TLSv1_2_server_method()**.

Then, the returned method (a pointer) can be used to create the context, e.g., for a client:

```
SSL_METHOD *method;
SSL_CTX *ctx;
method = TLSv1_2_client_method();
ctx = SSL_CTX_new(method);
```

Further settings regarding the context are required, namely loading a local public key certificate and corresponding private key from local files.

This is required if the authentication method is based on public key certificates and thus corresponding cypher suites are to be supported. The local certificate is to be presented to counterparts during the TLS handshake (Hello messages) as part of the negotiation.

In additional, and optionally, it's possible to restrict SSL/TLS versions and cypher suited to be supported .

OpenSSL – the local certificate

To authenticate both the client and the server, both are required to have a public key certificate and the corresponding private key.

The server is always required to have one, for the client it's optional, however, the server may demand a client's certificate. The local certificate and the corresponding private key are loaded from a file into the context by calling SSL_CTX_use_certificate_file() and SSL_CTX_use_PrivateKey_file() functions.

```
ctx = SSL_CTX_new(method);
SSL_CTX_use_certificate_file(ctx, filename1, SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, filename2, SSL_FILETYPE_PEM);
```

In this case, the PEM format is being used, with this format, both the certificate and the private key may be stored on the same file, if so, filename2 would be the same as filename1.

It should be checked if the loaded private key matches the loaded certificate, this can be accomplished by calling the **SSL_CTX_check_private_key(ctx)** function, if they don't match, zero will be returned.

OpenSSL – preparing the negotiation (optional)

Things are now prepared for the SSL/TLS negotiation; however, it may be wise to restrict negotiation options. The negotiation establishes the SSL/TLS version to be used, and the cypher suite to be used.

To set the minimal SSL/TLS version, for the created context, the following function can be used:

SSL_CTX_set_min_proto_version(SSL_CTX *ctx, int version);

Where version can be zero, standing for the lowest available version, or one of the following values, from lower to higher versions: SSL3_VERSION, TLS1_VERSION, TLS1_1_VERSION, TLS1_2_VERSION.

TLSv1 is the same as SSLv3.1, starting from SSLv3.1, SSL was renamed to TLS.

Regarding cypher suites, by default there's a significant number available to be used on the negotiation, they are included on the Client Hello message sent to the server. The list of supported cypher suites may be reduced by using the following function:

SSL_CTX_set_cipher_list(SSL_CTX *ctx, const char *str);

The provided string (str) has a special matching format, to include and exclude cypher suites, based on their names, and features, like security level.

OpenSSL – trusted entities (certificates)

Later, during the TLS handshake, the application is going to receive a public key certificate from the counterpart, this is always the case for the client. The server may or not demand the client's public key certificate.

When the certificate is received, it must be checked if it's valid, including if it was issued by a trusted entity, possibly a trusted CA. For the issuer to be trusted, it's required be in certification chain leading to a certificate regarded as trusted by the application.

Such a list of trusted certificates must be available and should be loaded into the context for later checking. The following function may be used:

```
SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char *CAfile,
                                            const char *CApath);
```

CAfile is a filename in PEM format holding a set of certificates, and CApath the name of a folder from where all existing files are loaded when verification is requested. CAfile may be NULL, in that case it's ignored, and the same goes for CApath. In Linux systems, the usual folders where trusted CA certificates are stored are /etc/ssl/certs/ or /etc/pki/tls/certs/. When using self-signed certificates, each application should load the counterpart's public key certificate for latter checking.

OpenSSL – requesting a certificate

As far as a public key certificates-based authentication cypher suite is selected, the default behaviour is the server sending a certificate to the client and not demanding a public key certificate from the client.

However, we may be interested in forcing the demand of a counterpart’s certificate, this will be the case for the server side, making it demand a client’s certificate on the Server Hello message. To demand a counterpart’s certificate and make the handshake fail if it fails to obey, the following function can be used:

```
SSL_CTX_set_verify(ctx, SL_VERIFY_PEER|SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
```

The last argument is a custom call-back function to check the received certificate during the handshake, if NULL, as above, the default library function will be used. This setting is mostly relevant on the server side, making the server include a demand for the client’s certificate on the Server Hello message. For a client, the effect is it will abort the handshake if the server’s certificate is not valid, otherwise the handshake is always successful.

Once the handshake is concluded, applications can then inspect the SSL/TLS attained connection and check if there’s a counterpart (peer) certificate, if it’s valid, and other properties, thus making required validations.

OpenSSL – securing the TCP connection

Until now, it has all been about preparing the context. Now things are ready to secure the TCP connection. From the created context, a new SSL connection must be created and associated with the connected socket. For a TCP client it's the socket attained after the successful call of the connect() function, for a TCP server it's the socket returned by the accept() function. As an example, imagining the connected socket is stored in a variable called **sok**, and the created context in a variable named **ctx**, then it would be:

```
SSL *sslConn;
sslConn= SSL_new(ctx);
SSL_set_fd(sslConn, sok);
```

The sslConn variable represents the SSL/TLS connection and is now associated with the connected socket. The SSL/TLS handshake can now start, the client calls **SSL_connect(sslConn)**, and the server calls **SSL_accept(sslConn)**. Both should return value one, otherwise it means the handshake has failed and the SSL/TLS connection has not been successfully established. Applications can now start exchanging information by reading and writing bytes through the SSL/TLS connection (sslConn). Once they are done, they should first close the SSL/TLS connection by calling **SSL_free(sslConn)**, and only then close the socket itself.

OpenSSL – sending and receiving bytes

Once the TCP connection is secured, data exchange can start.

Instead of using the read() function to read bytes from the socket, and the write() function to write bytes into the socket, now these functions are replaced by SSL_read() and SSL_write():

```
int SSL_read(SSL *ssl, void *buf, int num);

int SSL_write(SSL *ssl, const void *buf, int num);
```

Instead of receiving a connected socket as first argument they receive the SSL/TLS connection, elsewhere they are similar to the standard read() and write(). If successful they return the number of byte read or written, a returned value below one indicates an error.

We have stepped into data exchange, but one vital stage may be missing. The counterpart's public key certificate checking. This is always required for the client, the server may or not receive a certificate from the client.

If the SSL_CTX_set_verify() function was used to enforce the SL_VERIFY_PEER flag, and the received certificate is invalid, then the handshake fails. But before starting data exchanges the received certificate should be checked.

OpenSSL – checking the peer certificate

The counterpart's certificate (peer's certificate) is actually checked during the handshake as established for the context with the SSL_CTX_set_verify() function, however, if SL_VERIFY_PEER was not enforced, even if it's invalid, the handshake is successful, and the SSL/TLS connection is established. This validation confronts the certificate's issuer against the list of trusted certificates established for the context trough the SSL_CTX_load_verify_locations() function.

To get the result of the certificate validation during the handshake the following function can be used afterwards:

SSL_get_verify_result(sslConn)

If the received certificate is valid, this function returns the value X509_V_OK. However, if the peer hasn't provided any certificate, it returns X509_V_OK as well. So, we should first check that by attaining the peer certificate:

X509* cert=SSL_get_peer_certificate(sslConn);

If there's no peer certificate it returns NULL, otherwise a pointer to the certificate is returned, further details regarding the certificate fields can then be inspected.

OpenSSL – checking the peer's name

Depending on the scenario, an application may want to check if the received certificate has a common name (CN) attribute on the subject field, matching the DNS name of the counterpart it’s expecting to be talking with. This is vital for a client talking with a public server over the internet, meaning the certificate was issued for that specific server and thus authenticates the server.

Once the certificate is attained (SSL_get_peer_certificate), to store the subject’s field in a string the following code can be used:

```
#define BUF_SIZE 500

char subject[BUF_SIZE];

X509_NAME_oneline(X509_get_subject_name(cert),subject,BUF_SIZE);
```

The common name is at the end of the subject string, started by “/CN=“, so we can get it by using:

```
char *cn=strstr(subject,”/CN=“);

cn+=4;
```

Then we can check if the string starting at **cn** matches (strcmp) the expected DNS host name.

Example – checking certificates of HTTPS servers

For the sake of applying what has been studied, next we will analyse a client application that establishes a TCP connection with standard HTTPS servers on the internet (port number 443) and then secures it with SSL/TLS, the objective is presenting information about the certificate provided by the server.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/ssl.h>
#include <openssl/x509.h>

#define BUF_SIZE 500
#define SERVER_PORT "443"

int main(int argc, char **argv) {
        int err, sock;
        char line[BUF_SIZE];
        struct addrinfo  req, *list;

        if(argc!=2) {
                puts("HTTPS server's DNS name is required as argument");
                exit(1);
                }
```

Example – checking certificates of HTTPS servers

Initially it’s a standard TCP client. In this case, only once the socket is connected, then the SSL/TLS context is settled:

```
bzero((char *)&req,sizeof(req));

req.ai_family = AF_UNSPEC;
req.ai_socktype = SOCK_STREAM;
err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
if(err) {
       printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1); }

sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
if(sock==-1) {
       perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

if(connect(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
       perror("Failed connect"); freeaddrinfo(list); close(sock); exit(1);}

puts("Connected (TCP)");
const SSL_METHOD *method=SSLv23_method();
SSL_CTX *ctx = SSL_CTX_new(method);

// ABORT ON HANDSHAKE IF CERTIFICATE UNTRUSTED
//SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,NULL);

// LOAD TRUSTED CA CERTIFICATES
SSL_CTX_load_verify_locations(ctx,NULL,"/etc/ssl/certs");

// Disable some cyphers and require HIGH, standing for 128-bits or above keys
SSL_CTX_set_cipher_list(ctx, "HIGH:!aNULL:!kRSA:!PSK:!SRP:!MD5:!RC4");
```

Example – checking certificates of HTTPS servers

Continue with the SSL/TLS context setup, and then start the handshake, if successful, start analysing the server’s public key certificate.

```
 // SOME SERVERS MAY NOT SUPPORT TLS1.2, so don’t uncomment this
 // SSL_CTX_set_min_proto_version(ctx,TLS1_2_VERSION);

SSL *sslConn = SSL_new(ctx);
SSL_set_fd(sslConn, sock);
if(SSL_connect(sslConn)!=1) {
        puts("TLS handshake error");
        SSL_free(sslConn);
        close(sock);
        exit(1);
}

 puts("Secured connection (SSL/TLS)");
 X509* cert=SSL_get_peer_certificate(sslConn);
 X509_free(cert);

if(cert==NULL) {
        puts("Sorry: no certificate provided by the server");
        SSL_free(sslConn);
        close(sock);
        exit(1);
}

X509_NAME_oneline(X509_get_subject_name(cert),line,BUF_SIZE);
printf("Server's certificate subject: %s\n",line);
```

Example – checking certificates of HTTPS servers

Check if the subject’s CN attribute matches the server’s DNS name.

```
char *cn=strstr(line,“/CN=");
if(cn==NULL) {
       puts("Server's certificate CN not found");
}
else {
       cn+=4;
       if(strcmp(cn,argv[1])) {
               puts("SECURITY WARNING: the server's certificate CN attribute doesn't match");
       }
       else
               printf("Server name (%s) matches.\n", cn);
}

X509_NAME_oneline(X509_get_issuer_name(cert),line,BUF_SIZE);
printf("Server's certificate issuer: %s\n",line);

printf("TLS version: %s\nCypher suite: %s\n",SSL_get_cipher_version(sslConn),SSL_get_cipher(sslConn));

if(SSL_get_verify_result(sslConn)!=X509_V_OK) {
       puts("Sorry: untrusted server certificate");
       SSL_free(sslConn);
       close(sock);
       exit(1);
}
puts("The certificate is trusted");

SSL_free(sslConn);
close(sock);
exit(0);
}
```

In a lab class, students will compile and test this client application by contacting public HTTPS servers and others on the internet.