CyaSSL SSL Tutorial

Release 2.0.0

December 08, 2011

Contents

The CyaSSL embedded SSL library can easily be integrated into your existing application or device to provide enhanced communication security through the addition of SSL and TLS. CyaSSL has been targeted at embedded and RTOS environments, and as such, offers a minimal footprint and fast speeds. Minimum build sizes for CyaSSL range between 30-100kB depending on the selected build options and platform being used.

Although CyaSSL is an embedded SSL library, it's full feature set makes it very functional in a desktop environment as well. It is generally very easy to compile on new platforms, and includes several abstraction layers, including ones for operating system, custom I/O, and standard C library. For a full list of features and supported platforms, see the product page: http://yassl.com/yaSSL/Products_cyassl.html.

The goal of this tutorial is to walk through the integration of SSL and TLS into a simple application. Hopefully the process of going through this tutorial will also lead to a better understanding of SSL in general. It will use CyaSSL with a simple echoserver and echoclient example to keep things as simple as possible while still demonstrating the general procedure of adding SSL support to an application. The echoserver and echoclient examples have been taken from the popular book titled *Unix Network Programming, Volume 1, 3rd Edition* by Richard Stevens, Bill Fenner, and Andrew Rudoff. If you would like to reference the base examples used from this book, they can be found on the following pages:

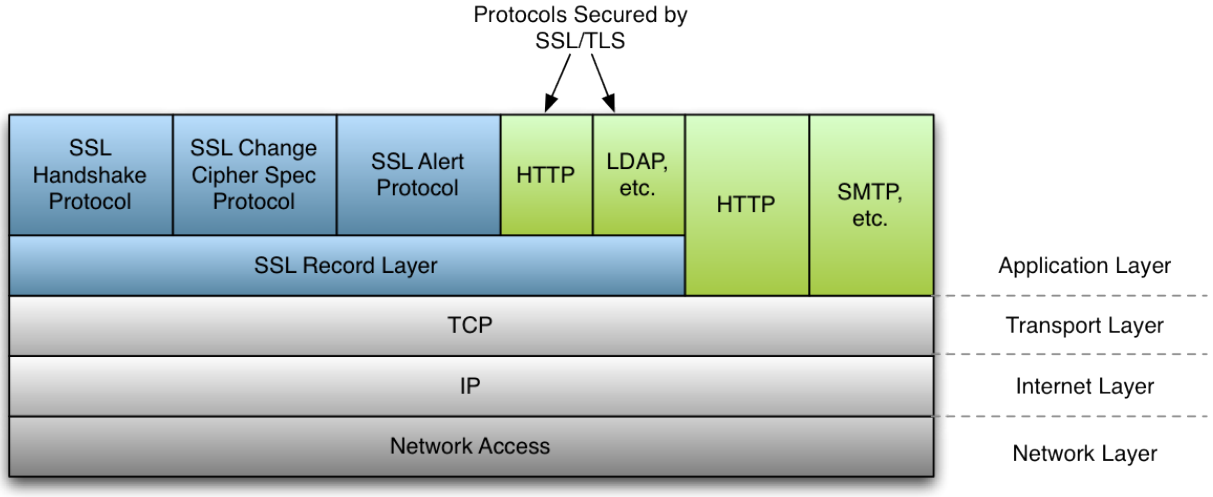
echoclient - Figure 5.4, Page 124
echoserver - Figure 5.12, Page 139

This tutorial assumes you are comfortable with editing and compiling C code using the GNU GCC compiler, as well as familiar with the concepts of public key encryption. Please note that access to the *Unix Network Programming* book is not required for this tutorial.

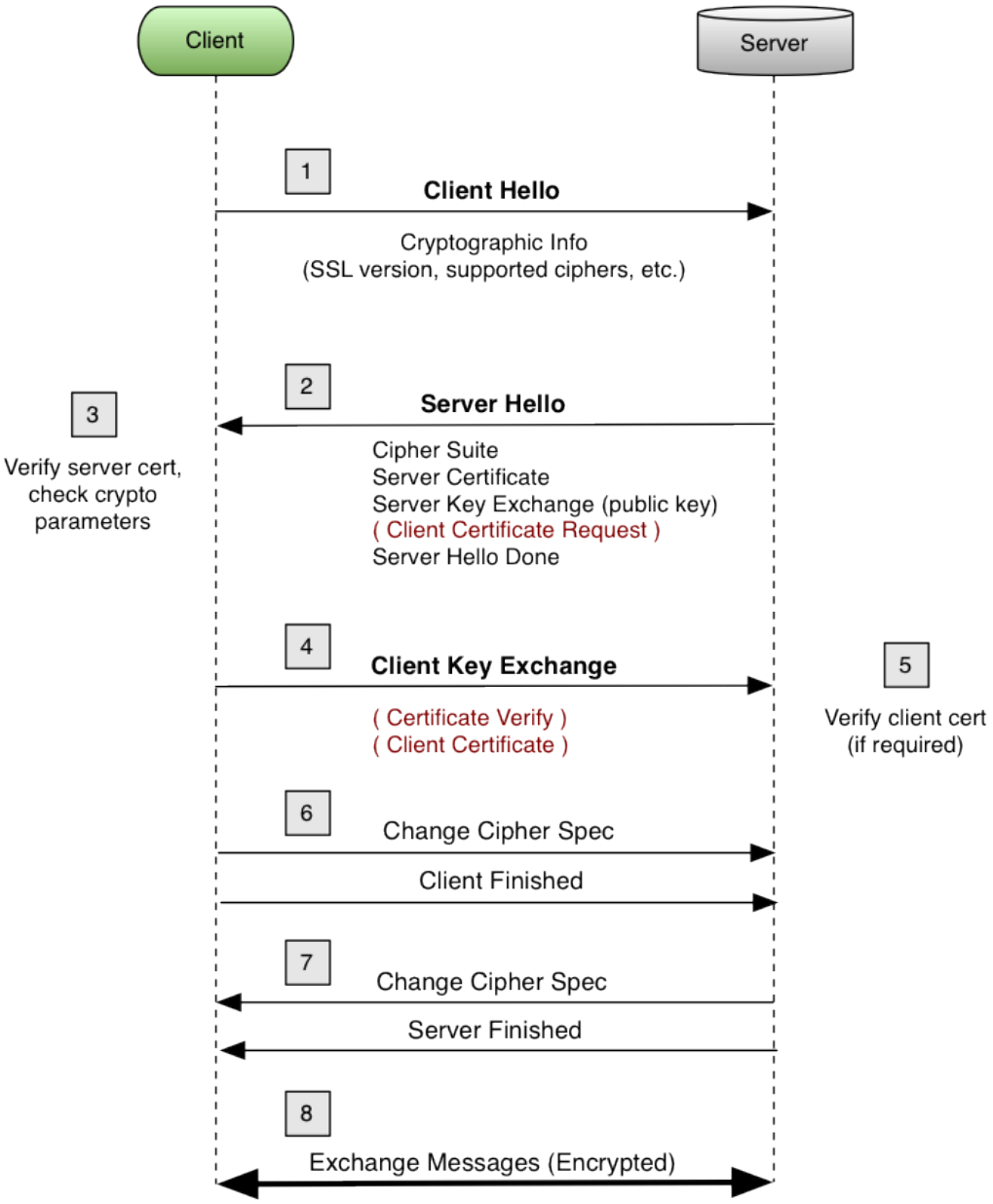
1 Quick Summary of SSL and TLS

TLS (Transport Layer Security) and **SSL** (Secure Sockets Layer) are cryptographic protocols that allow for secure communication across a number of different transport protocols - mainly TCP/IP. The most recent version of SSL/TLS is TLS 1.2. CyaSSL supports SSL 3.0, TLS 1.0, 1.1, and 1.2.

SSL and TLS sit between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the underlying transport medium. Application protocols are layered on top of SSL and can include protocols such as HTTP, FTP, and SMTP. A diagram of how SSL fits into the OSI model can be seen in the following figure.



During connection, SSL or TLS negotiates a certain subset of ciphers and certificates to use during the connection. The SSL handshake involves several steps, some of which are optional depending on what options the SSL client and server have been configured with. A simplified diagram of the SSL handshake can be seen in the following figure.



For more information about the history and details of SSL and TLS, please see either the Wikipedia page or the respective RFC document.

Wikipedia: TLS http://en.wikipedia.org/wiki/Transport_Layer_Security
SSL v3.0 http://tools.ietf.org/id/draft-ietf-tls-ssl-version3-00.txt
TLS v1.0 http://www.ietf.org/rfc/rfc2246.txt
TLS v1.1 http://www.ietf.org/rfc/rfc4346.txt
TLS v1.2 http://www.ietf.org/rfc/rfc5246.txt

2 Getting the Source Code

All of the source code used in this tutorial can be downloaded from the yaSSL website, specifically from the following location. The download contains both the original and completed source code for both the echoserver and echoclient used in this tutorial. Specific contents are listed below the link.

http://www.yassl.com/documentation/ssl-tutorial-2.0.zip

The downloaded ZIP file has the following structure:

```
CyaSSL_SSL_Tutorial.pdf
/finished_src
   /echoclient
      (The completed echoclient code)
   /echoserver
      (The completed echoserver code)
   /include
      (Common header file [Modified from unp.h in the book])
   /lib
      (Common library functions)
/original_src
   /echoclient
      (The starting echoclient code)
   /echoserver
      (The starting echoserver code)
   /include
      (Common header file [Modified from unp.h in the book])
   /lib
      (Common library functions)
```

3 Base Example Modifications

This tutorial, and the source code that accompanies it, have been designed to be as portable as possible across platforms. Because of this, and because we want to focus on how to add SSL and TLS into an application, the base examples have been kept as simple as possible. Several modifications have been made to the examples taken from *Unix Network Programming* in order to either remove unnecessary complexity or increase the range of platforms supported. If you believe there is something we could do to increase the portability of this tutorial, please let us know at support@yassl.com.

The following is a list of modifications that were made to the echoserver and echoclient examples.

Modifications to the echoserver (tcpserv04.c)

- Removed call to the Fork() function because fork() is not supported by Windows. The result of this is an echoserver which only accepts one client simultaneously. Along with this removal, Signal handling was removed.
- Moved str_echo() function from str_echo.c file into tcpserv04.c file
- Added a printf statement to view the client address and the port we have connected through:

```
printf("Connection from %s, port %d\n",
   inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
   ntohs(cliaddr.sin_port));
```

- Added a call to `setsockopt()` after creating the listening socket to eliminate the "Address already in use" bind error.

Modifications to the echoclient (tcpcli01.c)

- Moved str_cli() function from str_cli.c file into tcpcli01.c file.

Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these source code examples, certain functions will be capitalized. For example, `Fputs()` and `Writen()`. The authors of *Unix Network Programming* have written custom wrapper functions for normal functions in order to cleanly handle error checking. For a more thorough explanation of this, please see **Section 1.4** (page 11) in the *Unix Network Programming* book.

4 Building and Installing CyaSSL

Before we begin, download the example code (echoserver and echoclient) from the **Getting the Source Code** section, above. This section will explain how to download, configure, and install the CyaSSL embedded SSL library on your system.

You will need to download and install the most recent version of CyaSSL from the yaSSL download page (http://yassl.com/yaSSL/Download.html). CyaSSL can be built with any number of available build options which allow you to enable or disable desired features such as DTLS, certificate generation, OpenSSL compatibility, and much more.

For a full list of available build options, see the "Building CyaSSL" guide (http://yassl.com/yaSSL/Docs_Building_CyaSSL.html). CyaSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building CyaSSL, please don't hesitate to ask for support through the yaSSL product support forums (http://www.yassl.com/forums).

When building CyaSSL on Linux, *BSD, OS X, Solaris, or other *nix like systems, you can use the autoconf system. To configure and build CyaSSL, run the following two commands from the terminal. Any desired build options may be appended to ./configure (ex: ./configure –enable-opensslExtra):

```
./configure
make
```

To install CyaSSL, run:

```
sudo make install
```

This will install CyaSSL headers into **/usr/local/include/cyassl** and the CyaSSL libraries into **/usr/local/lib** on your system. To test the build, run the testsuite application from the CyaSSL root directory:

```
./testsuite/testsuite
```

A set of tests will be run on CTaoCrypt and CyaSSL to verify it has been installed correctly. After a successful run of the `testsuite` application, you should see output similar to the following:

```
MD5      test passed!
MD4      test passed!
SHA      test passed!
SHA-256  test passed!
HMAC     test passed!
ARC4     test passed!
Rabbit   test passed!
DES      test passed!
DES3     test passed!
AES      test passed!
RANDOM   test passed!
```

```
RSA      test passed!
DH       test passed!
DSA      test passed!
PWDBASED test passed!
OPENSSL  test passed!
peer's cert info:
 issuer : /C=US/ST=Oregon/L=Portland/O=yaSSL/OU=programming/CN=www.yassl.com/emailAddress=info@yassl
 subject: /C=US/ST=Oregon/L=Portland/O=yaSSL/OU=programming/CN=www.yassl.com/emailAddress=info@yassl
 serial number:c5:d7:6c:11:36:f0:35:e1
SSL version is TLSv1.2
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
peer's cert info:
 issuer : /C=US/ST=Montana/L=Bozeman/O=sawtooth/OU=consulting/CN=www.sawtooth-consulting.com/emailAdd
 subject: /C=US/ST=Montana/L=Bozeman/O=yaSSL/OU=support/CN=www.yassl.com/emailAddress=info@yassl.com
 serial number:01
SSL version is TLSv1.2
SSL cipher suite is TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
Client message: hello cyassl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
b88596cd2362310b2506f9d73693cefd  input
b88596cd2362310b2506f9d73693cefd  output

All tests passed!
```

Now that CyaSSL has been installed, we can begin modifying the example code to add SSL functionality. We will first begin by adding SSL to the echoclient and subsequently move on to the echoserver.

5 Initial Compilation

To compile and run the example echoclient and echoserver code from **ssl_tutorial.zip**, you can use the included Makefiles. Change directory (cd) to either the echoclient or echoserver directory and run:

```
make
```

This will compile the example code and produce an executable named either `echoserver` or `echoclient` depending on which one is being built. The GCC command which is used in the Makefile can be seen below. If you want to build one of the examples without using the supplied Makefile, change directory to the example directory and replace **tcpserv04.c** in the following command with correct source file for the example:

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

This will compile the current example into an executable, creating either an "echoserver" or "echoclient" application. To run one of the examples after it has been compiled, change your current directory to the desired example directory and start the application. For example, to start the `echoserver` use:

```
./echoserver
```

When running the `echoclient` you will need to supply the IP address of the server when staring the application, which in our case will be **127.0.0.1**. Change your current directory to the "echoclient" directory and run the following command. Note that the `echoserver` must already be running:

```
./echoclient 127.0.0.1
```

Once you have both the `echoserver` and `echoclient` running, the echoserver should echo back any input that it receives from the echoclient. To exit either the echoserver or echoclient, use **[Ctrl + C]** to quit the application.

Currently, the data being echoed back and forth between these two examples is being sent in the clear - easily allowing anyone with a little bit of skill to inject themselves in between the client and server and listen to your communication.

6 Libraries

The CyaSSL library, once compiled, is named libcyassl, and unless otherwise configured the CyaSSL build and install process creates only a shared library under the following directory. Both shared and static libraries may be enabled or disabled by using the appropriate build options.:

```
/usr/local/lib
```

The first step we need to do is link the CyaSSL library to our example applications. Modifying the GCC command (using the `echoserver` as an example), gives us the following new command. Since CyaSSL installs header files and libraries in standard locations, GCC should be able to find them without explicit instructions (using -l or -L). Note that by using **-lcyassl** the compiler will automatically choose the correct type of library (static or shared):

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lcyassl
```

7 Headers

Now that the echoclient and echoserver applications have been compiled and linked to the CyaSSL library, we can modify the source code of the example applications. We're going to look at the `echoclient` first, then move on to the `echoserver`. The first thing we will need to do is include the CyaSSL OpenSSL compatibility header. Open the `tcpcli01.c` file and add the following line near the top:

```
#include <cyassl/ssl.h>
```

8 Startup/Shutdown

Before we can use CyaSSL in our code, we need to initialize the library and the CYASSL_CTX. CyaSSL is initialized by calling `CyaSSL_Init()`. This must be done first before anything else can be done with the library.

The **CYASSL_CTX** structure (CyaSSL Context) contains global values for each SSL connection, including certificate information. A single CYASSL_CTX can be used with any number of CYASSL objects created. This allows us to load certain information, such as a list of trusted CA certificates only once.

To create a new CYASSL_CTX, use `CyaSSL_CTX_new()`. This function requires an argument which defines the SSL or TLS protocol for the client to use. There are several options for selecting the desired protocol. CyaSSL currently supports SSLv3, TLSv1, TLSv1.1, TLSv1.2, and DTLS. Each of these protocols have a corresponding function that can be used as an argument to `CyaSSL_CTX_new()`. The possible client protocol options are shown below. SSL 2.0 is not supported by CyaSSL because it has been insecure for several years:

```
CyaSSLv3_client_method();     // SSL 3
CyaTLSv1_client_method();     // TLS 1
CyaTLSv1_1_client_method();   // TLS 1.1
CyaTLSv1_2_client_method();   // TLS 1.2
CyaSSLv23_client_method();    // Use highest version possible from SSLv3 - TLS 1.2
CyaDTLSv1_client_method();    // DTLS
```

We need to load our CA (Certificate Authority) certificate into the CYASSL_CTX so that the when the echoclient connects to the echoserver, it is able to verify the server's identity. To load the CA certificates into the CYASSL_CTX, use `CyaSSL_CTX_load_verify_locations()`. This function requires three arguments: a CYASSL_CTX

pointer, a certificate file, and a path value. The **path** value points to a directory which should contain CA certificates in PEM format. When looking up certificates, CyaSSL will look at the **certificate file** value before looking in the **path** location. In this case, we don't need to specify a certificate path because we will specify one CA file - as such we use the value 0 for the path argument. The `CyaSSL_CTX_load_verify_locations` function returns either `SSL_SUCCESS` or `SSL_FAILURE`:

```
CyaSSL_CTX_load_verify_locations(CYASSL_CTX* ctx, const char* file, const char* path)
```

Putting these things together (library initialization, protocol selection, and CA certificate), we have the following. Here, we choose to use TLS 1.0:

```
CyaSSL_Init();       // Initialize CyaSSL
CYASSL_CTX* ctx;

/* Create the CYASSL_CTX */
if ( (ctx = CyaSSL_CTX_new(CyaTLSv1_client_method())) == NULL){
   fprintf(stderr, "CyaSSL_CTX_new error.\n");
   exit(EXIT_FAILURE);
}

/* Load CA certificates into SSL_CTX */
if (CyaSSL_CTX_load_verify_locations(ctx,"./ca-cert.pem",0) != SSL_SUCCESS) {
   fprintf(stderr, "Error loading ./ca-cert.pem, please check the file.\n");
   exit(EXIT_FAILURE);
}
```

The code shown above should be added to the beginning of **tcpcli01.c**, after the variable definitions and the check that the user has started the client with an IP address. A version of the completed code is included in the **ssl_tutorial.zip** file for reference.

Now that CyaSSL and the CYASSL_CTX have been initialized, make sure that the CYASSL_CTX object and the CyaSSL library are freed when the application is completely done using SSL. The following two lines should be placed at the end of the echoclient's `main()` function - right before the call to `exit(0)`:

```
CyaSSL_CTX_free(ctx);
CyaSSL_Cleanup();
```

9 CYASSL Object

A CYASSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session. In the echoclient example, we will do this after the call to `Connect()`, shown below:

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

Create a new CYASSL object using the `CyaSSL_new()` function. This function returns a pointer to the CYASSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (**sockfd**) with the new CYASSL object (**ssl**):

```
/* Create CYASSL object */
CYASSL* ssl;

if( (ssl = CyaSSL_new(ctx)) == NULL) {
   fprintf(stderr, "CyaSSL_new error.\n");
   exit(EXIT_FAILURE);
}
```

```
CyaSSL_set_fd(ssl, sockfd);
```

One thing to notice here is we haven't made a call to the `CyaSSL_connect()` function. `CyaSSL_connect()` initiates the SSL/TLS handshake with the server, and is called during `CyaSSL_read()` if it hasn't been called previously. In our case, we don't explicitly call `CyaSSL_connect()`, as we let our first `CyaSSL_read()` do it for us.

10 Sending Data

The next step is to begin sending data securely. The echoclient example uses the functions `Writen()` and `Readline()` to send and receive data between it and the echoserver. These calls need to be replaced with calls to CyaSSL's `CyaSSL_write()` and `CyaSSL_read()` functions.

Take note that in the echoclient example, the `main()` function hands off the sending and receiving work to `str_cli()`. The `str_cli()` function is where our function replacements will be made. First we need access to our SSL object in the `str_cli()` function, so we add another argument and pass the **ssl** variable to `str_cli()`. Because the **CYASSL** object is now going to be used inside of the `str_cli()` function, we remove the **sockfd** parameter. The new `str_cli()` function signature after this modification is shown below:

```
void
str_cli(FILE *fp, CYASSL* ssl)
```

In the `main()` function, the new argument (ssl) is passed to `str_cli()`:

```
str_cli(stdin, ssl);
```

Inside the `str_cli()` function, `Writen()` and `Readline()` are replaced with CyaSSL functions, and the CYASSL object (ssl) is used instead of the original file descriptor(sockfd). The new `str_cli()` function is shown below. Notice that we now need to check if our calls to `CyaSSL_write` and `CyaSSL_read` were successful.

The authors of the Unix Programming book wrote error checking into their `Writen()` function which we must make up for after it has been replaced. We add a new `int` variable, "n", to monitor the return value of `CyaSSL_read` and before printing out the contents of the buffer, `recvline`, the end of our read data is marked with a '0':

```
void
str_cli(FILE *fp, CYASSL* ssl)
{
   char      sendline[MAXLINE], recvline[MAXLINE];
   int    n = 0;

   while (Fgets(sendline, MAXLINE, fp) != NULL) {

   if(CyaSSL_write(ssl, sendline, strlen(sendline)) != strlen(sendline)){
      err_sys("CyaSSL_write failed");
   }

   if ((n = CyaSSL_read(ssl, recvline, MAXLINE)) <= 0)
      err_quit("CyaSSL_read error");

      recvline[n] = '\0';
      Fputs(recvline, stdout);
   }
}
```

The last thing to do is free the CYASSL object when we are completely done with it. In the `main()` function, right before the line to free the CYASSL_CTX, call to `CyaSSL_free()`:

```
str_cli(stdin, ssl);

CyaSSL_free(ssl);               // Free SSL object
CyaSSL_CTX_free(ctx);    // Free SSL_CTX object
CyaSSL_Cleanup();               // Free CyaSSL
```

11 Signal Handling

There is a strong possibility that a user will close the echoclient by using "Ctrl+C". In order for CyaSSL resources to be released, this signal should be caught in order to handle the program exit gracefully. There are two things which we will do:

- Add a signal handler function (here, we added it before the `str_cli()` function):

```
void sig_handler(const int sig)
{
   printf("\nSIGINT handled.\n");
   CyaSSL_Cleanup();                  /* Free CyaSSL */
   exit(EXIT_SUCCESS);
}
```

- Register this function as a signal handler using the `signal()` function. We added this directly after variable declarations in the `main()` method of the `echoclient`:

```
/* define a signal handler for when the user closes the program with Ctrl-C */
signal(SIGINT, sig_handler);
```

That's it - the echoclient is now enabled with TLSv1!! We included the CyaSSL headers, initialized CyaSSL, created an CYASSL_CTX structure in which we chose what protocol we wanted to use, created an CYASSL object to use for sending and receiving data, replaced calls to `Writen()` and `Readline()` with `CyaSSL_write()` and `CyaSSL_read()`, freed CYASSL, CYASSL_CTX, and CyaSSL, and then made sure we handled the Ctrl+C signal.

There are many more aspects and methods to configure and control the behavior of your SSL connections. For more detailed information, please see additional CyaSSL documentation and resources. The next section will deal with enabling TLSv1 in the echoserver example.

12 Echo Server

Enabling SSL/TLS in the echoserver example is very similar to the steps above for the echoclient. Follow the steps above, except when choosing the protocol version (during the creation of the CYASSL_CTX structure in the **Startup/Shutdown** section, above), we must use a server method instead. There are several options which may be chosen for the server protocol:

```
CyaSSLv3_server_methods();    // SSLv3
CyaTLSv1_server_method();     // TLSv1
CyaTLSv1_1_server_method();   // TLSv1.1
CyaTLSv1_2_server_method();   // TLSv1.2
CyaSSLv23_server_method();    // Allow clients to connect with SSLv3 or TLSv1+
CyaDTLSv1_server_method();    // DTLS
```

The resulting call to `CyaSSL_CTX_new()` should be similar to this:

```
/* Create and initialize SSL_CTX structure */
if ( (ctx = CyaSSL_CTX_new(CyaTLSv1_server_method())) == NULL){
```

```
        fprintf(stderr, "CyaSSL_CTX_new error.\n");
        exit(EXIT_FAILURE);
}
```

When loading certificates into the CYASSL_CTX, the server certificate and key file should be loaded in addition to the CA certificate. This will allow the server to send the client its certificate for identification verification:

```
// Load server certificate
if (CyaSSL_CTX_use_certificate_file(ctx,"./server-cert.pem", SSL_FILETYPE_PEM) != SSL_SUCCESS) {
   fprintf(stderr, "Error loading ./server-cert.pem, please check the file.\n");
   exit(EXIT_FAILURE);
}

// Load server private key
if (CyaSSL_CTX_use_PrivateKey_file(ctx,"./server-key.pem", SSL_FILETYPE_PEM) != SSL_SUCCESS) {
   fprintf(stderr, "Error loading ./server-key.pem, please check the file.\n");
   exit(EXIT_FAILURE);
}
```

The echo server makes a call to `str_echo()` to handle reading and writing (whereas the client made a call to `str_cli()`). As with the client, modify `str_echo()` by replacing the **sockfd** parameter with an CYASSL object (**ssl**) parameter in the function signature:

```
void str_echo(CYASSL* ssl)
```

Replace the calls to `read()` and `Writen()` with calls to the `CyaSSL_read()` and `CyaSSL_write()` functions. The modified `str_echo()` function, including error checking of return values, is shown below. Note that the type of the variable "n" has been changed from `ssize_t` to `int` in order to accommodate for the change from `read()` to `SSL_read()`:

```
void
str_echo(CYASSL* ssl)
{
   int                        n;
   char              buf[MAXLINE];

again:
   while ( (n = CyaSSL_read(ssl, buf, MAXLINE)) > 0) {
      if(CyaSSL_write(ssl, buf, n) != n) {
         err_sys("CyaSSL_write failed");
      }
   }

   if( n < 0 )
      printf("CyaSSL_read error = %d\n", CyaSSL_get_error(ssl,n));

   else if( n == 0 )
      printf("The peer has closed the connection.\n");
}
```

Like the `echoclient`, we will need to add a signal handler for when the user closes the `echoserver` by using "Ctrl+C". The echo server is continually running in a loop. Because of this, we need to provide a way to break that loop when the user presses "Ctrl+C". To do this, the first thing we need to do is change our loop to a `while` loop which terminates when an exit variable (`cleanup`) is set to true.

First, define a new static int variable called `cleanup` at the top of `tcpserv04.c` right after the #include statements:

```
static int cleanup;          // To handle shutdown
```

Modify the `echoserver` loop by changing it from a `for` loop to a `while` loop:

```
while(cleanup != 1)
{
   // echo server code here
}
```

For the `echoserver` we need to disable the operating system from restarting calls which were being executed before the signal was handled after our handler has finished. By disabling these, the operating system will not restart calls to accept() after the signal has been handled. If we didn't do this, we would have to wait for another client to connect and disconnect before the echoserver would clean up resources and exit.

To define the signal handler and turn off SA_RESTART, first define `act` and `oact` structures in the echoserver's `main` function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to `CyaSSL_Init()` in the main function:

```
/* Define a signal handler for when the user closes the program with Ctrl-C
Also, turn off SA_RESTART so that the OS doesn't restart the call to accept()
after the signal is handled. */

act.sa_handler = sig_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGINT, &act, &oact);
```

The echoserver's `sig_handler` function is shown below:

```
void sig_handler(const int sig)
{
   printf("\nSIGINT handled.\n");
       cleanup = 1;
       return;
}
```

Once again, the completed source code can be found in the downloaded ZIP file.

13 Certificates

For testing purposes, you may use the certificates provided by CyaSSL. These can be found in the CyaSSL download, and specifically for this tutorial, they can be found in the **finished_src** folder.

For production applications, you should obtain correct and legitimate certificates from a trusted certificate authority.

14 Conclusion

This tutorial walked through the process of integrating the CyaSSL embedded SSL library into a simple client and server application. Although this example is simple, the same principles may be applied for adding SSL or TLS into your own application. The CyaSSL embedded SSL library provides all the features you would need in a compact and efficient package that has been optimized for both size and speed.

Being dual licensed under GPLv2 and standard commercial licensing, you are free to download the CyaSSL source code directly from our website. Feel free to post to our support forums (www.yassl.com/forums) with any questions or comments you might have. If you would like more information about our products, please contact info@yassl.com.

We welcome any feedback you have on this SSL tutorial. If you believe it could be improved or enhanced in order to make it either more useful, easier to understand, or more portable, please let us know at support@yassl.com.