http://www.yassl.com
mailto:info@yassl.com
phone: +1 206 369 4800

Getting Started with CyaSSL

General Description

CyaSSL is a C language based SSL developed for embedded environments and real time operating systems where resources are constrained. CyaSSL is about 10 times smaller than yaSSL and up to 20 times smaller than OpenSSL when using the compile options described below. User benchmarking and feedback also reports dramatically better performance from CyaSSL vs. OpenSSL in the vast majority of standard SSL operations.

For instructions on the build process please see the document "*Building CyaSSL*".

Testsuite

The *testsuite* program is designed test the ability of CyaSSL and its cryptography library CTaoCrypt to run on the system. On a successful run you should see output like:

```
MD5             test passed!
MD4             test passed!
SHA             test passed!
SHA-256         test passed!
HMAC            test passed!
ARC4            test passed!
HC-128          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!
OPENSSL         test passed!
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=yaSSL/CN=www.yassl.com/
emailAddress=info@yassl.com
peer's cert info:
issuer : /C=US/ST=Oregon/L=Portland/O=sawtooth/CN=www.sawtooth-
consulting.com/emailAddress=info@yassl.com
subject: /C=US/ST=Oregon/L=Portland/O=taoSoftDev/
CN=www.taosoftdev.com/emailAddress=info@yassl.com
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!
```

This indicates that everything is configured and built correctly. If any of the tests fail make sure the build system was set up correctly. Likely culprits include having the wrong endianness or not properly setting the 64 bit type. If you've set anything to the non-default settings try removing those and rebuilding, retesting.

Client example

You can use the client example found in examples/client to test CyaSSL against any SSL server. To test against secure gmail try:

```
./client gmail.google.com 443
peer's cert info:
issuer : /C=US/O=Google Inc/CN=Google Internet Authority
subject: /C=US/ST=California/L=Mountain View/O=Google Inc/
CN=*.google.com
SSL connect ok, sending GET...
Server response: HTTP/1.0 302 Found
Cache-Control: private
Location: http://www.google.com
Content-Type: text/html; charset=UTF-8
Content-Length: 218
Date: Tue, 16 Feb 2010 22:25:02 GMT
Server: GFE/2.0
X-XSS-Protection: 0
```

This tells the client to connect to gmail.google.com on the https port of 443 and sends a generic GET. The rest is the initial output from the server that fits into the read buffer.

If no command line arguments are given then the client attempts to connect to the localhost on the yassl default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

If one command line argument is given the client attempts to connect to the localhost at port 11111 the argument number of times and gives the average time in milliseconds that it took to perform SSL_connect(). For example,

```
./client 100
SSL_connect avg took: 0.653 milliseconds
```

If you'd like to change the default host from localhost or the default port from 11111 you can change these settings in *test.h* located in /examples. The variables **yasslIP** and **yasslPort** control these settings. Rebuild all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to eachother.

Server example

The server example demonstrates a simple SSL server that performs client authentication and fails if the client doesn't present a certificate. Only one client connection is accepted and then the server quits. The client example in normal mode (no command line arguments) will work just fine against the example server. But if you specify command line arguments for the client example then a client certificate isn't loaded and the *SSL_connect()* will fail. The server will report an error "**-245, peer didn't send cert**".

EchoServer example

The echoserver example sits in an endless loop waiting for an unlimited number of client connections. Whatever the client sends the echoserver echos back. Client authentication isn't performed so the example client can be used against the echoserver in all 3 modes. Four special commands aren't echoed back and instruct the echoserver to take a different action.

1) "**quit**" If the echoserver receives the string "quit" it will shutdown.

2) "**break**" If the echoserver receives the string "break" it will stop the current session but continue handling requests. This is particuluary useful for DTLS testing.

3) "**printstats**" If the echoserver receives the string "printstats" it will print out statistics for the session cache.

4) "**GET**" If the echoserver receives the string "GET" it will handle it as an http get and send back a simple page with the message "greeting from CyaSSL". This allows testing of various TLS/SSL clients like Safari, IE, Firefox, gnutls, and the like against the echoserver example.

The output of echoserver is echoed to **stdout** unless **NO_MAIN_DRIVER** is defined. You can redirect output through the shell or through the first command line argument. To create a file named output.txt with the output from the echoserver run:

```
./echoserver outupt.txt
```

EchoClient example

The echoclient example can be run in interactive mode or batch mode with files. To run in interactive mode and write 3 strings "hello", "cyassl", and "quit" results in:

```
./echoclient
hello
hello
cyassl
cyassl
quit
sending server shutdown command: quit!
```

To use an input file specify the file name on the command line as the first argument. So to echo the contents of the file input.txt issue:

```
./echoclient input.txt
```

If you want the result to be written out to a file you can specify the output file name as an additional command line argument. The following command will echo the contents of file input.txt and write the result from the server to output.txt

```
./echoclient input.txt output.txt
```

The testsuite program does just that but hashes the input and output files to make sure that the client and server were getting/sending the correct and expected results.

Benchmark

The benchmark utility located in ctaocrypt/benchmark can be used to benchmark the cryptographic functionality of CTaoCrypt. Typical output may look like:

```
./benchmark
AES         5 megs took 0.043 seconds, 116.50 MB/s
ARC4        5 megs took 0.026 seconds, 194.72 MB/s
HC128       5 megs took 0.006 seconds, 901.07 MB/s
RABBIT      5 megs took 0.017 seconds, 299.11 MB/s
3DES        5 megs took 0.284 seconds,  17.62 MB/s

MD5         5 megs took 0.015 seconds, 334.59 MB/s
SHA         5 megs took 0.031 seconds, 163.16 MB/s
SHA-256     5 megs took 0.052 seconds,  96.28 MB/s

RSA 1024 encryption took 0.06 milliseconds, avg
RSA 1024 decryption took 0.61 milliseconds, avg
DH  1024 key generation  0.25 milliseconds, avg
DH  1024 key agreement   0.27 milliseconds, avg
```

This is especially useful for comparing the public key speed before and after changing the math library. You can test the results using the normal math library, the fastmath library, and the fasthugemath library.

Changing a client application to use CyaSSL

1) Include the CyaSSL OpenSSL compatibility header

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

2) Change all calls from *read()* (or r*ecv()*) to *SSL_read()* so

```
result = read(fd, buffer, bytes);
```

becomes

```
result = SSL_read(ssl, buffer, bytes);
```

3) Change all calls from write (or send) to *SSL_write()* so

```
result = wirte(fd, buffer, bytes);
```

becomes

```
result = SSL_wirte(ssl, buffer, bytes);
```

4) You can manually call *SSL_connect()* but that's not even necessary, the first call to *SSL_read()* or *SSL_write()* will initiate the *SSL_connect()* if it hasn't taken place yet.

5) Initialize CyaSSL and the SSL_CTX. You can use one SSL_CTX no matter how many SSL ojbects you end up creating. Basically you'll just have to load CA certificates to verify the server you're connecting to. Basic initialization looks like:

```
InitCyaSSL();

SSL_CTX* ctx;

if ( (ctx = SSL_CTX_new(TLSv1_client_method())) == NULL) {
    fprintf(stderr, "SSL_CTX_new error.\n");
    exit(EXIT_FAILURE);
}

if (SSL_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);
}
```

6) Create the SSL object after each tcp connect and associate the file descriptor with the session:

```
// after connecting to socket fd

SSL* ssl;

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

SSL_set_fd(ssl, fd);
```

7) Error checking. Each *SSL_read() SSL_write()* call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like *read()* and *write()*. In the event of an error you can use two calls to get more information about the error:

```
char errorString[80];
```

```
int err = SSL_get_error(ssl, 0);
ERR_error_string(err, buffer);
```

If you are using non blocking sockets you can test for errno **EAGAIN/EWOULDBLOCK** or more correctly you can test the specific error code for **SSL_ERROR_WANT_READ** or **SSL_ERROR_WANT_WRITE**.

8) Cleanup. After each SSL object is done being used you can free it up by calling:

```
SSL_free(ssl);
```

When you are completely done using SSL altogether you can free the SSL_CTX object by calling:

```
SSL_CTX_free(ctx);
FressCyassl();
```

Changing a server application to use CyaSSL

1) Follow the instructions above for a client except change the client method call in step 5 to a server one, so

```
SSL_CTX_new(TLSv1_client_method())
```

becomes

```
SSL_CTX_new(TLSv1_server_method())
```

or even

```
SSL_CTX_new(SSLv23_server_method())
```

To allow SSLv3 and TLSv1+ clients to connect to the server.

2) Add the server's certificate and key file to the initialization in step 5 above:

```
if (SSL_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);
}
```

```
if (SSL_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);
}
```