An Introduction to OpenSSL Programming, Part I of II

Sep 01, 2001 By Eric Rescorla (/user/800894)
in

Do you have a burning need to build a simple web client and server pair? Here's why OpenSSL is for you.

(/issue/89)

From Issue #89
September 2001 (/issue/89)

The Client

Once the client has initialized the SSL context, it's ready to connect to the server. OpenSSL requires us to create a TCP connection between client and server on our own and then use the TCP socket to create an SSL socket. For convenience, we've isolated the creation of the TCP connection to the tcp_connect() function (which is not shown here but is available in the downloadable source).

Once the TCP connection has been created, we create an SSL object to handle the connection. This object needs to be attached to the socket. Note that we don't directly attach the SSL object to the socket. Rather, we create a BIO object using the socket and then attach the SSL object to the BIO.

This abstraction layer allows you to use OpenSSL over channels other than sockets, provided you have an appropriate BIO. For instance, one of the OpenSSL test programs connects an SSL client and server purely through memory buffers. A more practical use would be to support some protocol that can't be accessed via sockets. For instance, you could run SSL over a serial line.

The first step in an SSL connection is to perform the SSL handshake. The handshake authenticates the server (and optionally the client) and establishes the keying material that will be used to protect the rest of the traffic. The SSL_connect() call is used to perform the SSL handshake. Because we're using blocking sockets, SSL_connect() will not return until the handshake is completed or an error has been detected. SSL_connect() returns 1 for success and 0 or negative for an error. This call is shown below:

```
/* Connect the TCP socket*/
sock=tcp_connect(host,port);
/* Connect the SSL socket */
ssl=SSL_new(ctx);
sbio=BIO_new_socket(sock,BIO_NOCLOSE);
SSL_set_bio(ssl,sbio,sbio);
if(SSL_connect(ssl)<=0)
  berr_exit("SSL connect error");
if(require_server_auth)
  check_cert(ssl,host);
```

When we initiate an SSL connection to a server, we need to check the server's certificate chain. OpenSSL does some of the checks for us, but unfortunately others are application specific, and so we have to do those ourselves. The primary test that our sample application does is to check the server identity. This check is performed by the check_cert function, shown in Listing 2.

Listing 2. check_cert() Function
(/files/linuxjournal.com/linuxjournal/articles/048/4822/4822l2.html)

Once you've established that the server's certificate chain is valid, you need to verify that the certificate you're looking at matches the identity that you expect the server to have. In most cases, this means that the server's DNS name appears in the certificate, either in the Common Name field of the Subject Name or in a certificate extension. Although each protocol has slightly different rules for verifying the server's identity, RFC 2818 contains the rules for HTTP over SSL/TLS. Following RFC 2818 is generally a good idea unless you have some explicit reason to do otherwise.

Because most certificates still contain the domain name in the Common Name field rather than in an extension, we show only the Common Name check. We simply extract the server's certificate using SSL_get_peer_certificate() and then compare the common name to the hostname we're connecting to. If they don't match, something is wrong and we exit.

Before version 0.9.5, OpenSSL was subject to a certificate extension attack. To understand this, consider the case where a server authenticates with a certificate signed by Bob, as shown in Figure 1. Bob isn't one of your CAs, but his certificate is signed by a CA you trust.

Figure 1. An Extended Certificate Chain

If you accept this certificate you're going to be in a lot of trouble. The fact that the CA signed Bob's certificate means that the CA believes that it has verified Bob's identity, not that Bob can be trusted. If you know that you want to do business with Bob, that's fine, but it's not very useful if you want to do business with Alice, and Bob (of whom you've never heard) is vouching for Alice.

Originally, the only way to protect yourself against this sort of attack was to restrict the length of certificate chains so that you knew that the certificate you were looking at was signed by the CA. The X.509 version 3 contains a way for a CA to label certain certificates as other CAs. This permits a CA to have a single root that then certifies a bunch of subsidiary CAs.

Modern versions of OpenSSL (0.9.5 and later) check these extensions, so you're automatically protected against extension attacks whether or not you check chain length. Versions prior to 0.9.5 do not check the extensions at all, so you have to enforce the chain length if using an older version. 0.9.5 has some problems with checking, so if you're using 0.9.5, you should probably upgrade. The #ifdefed code in initialize_ctx() provides chain-length checking with older versions. We use the SSL_CTX_set_verify_depth() to force OpenSSL to check the chain length. In summary, it's highly advisable to upgrade to 0.9.6, particularly because longer (but properly constructed) chains are becoming more popular. The absolute latest (and best) version of OpenSSL is .0.9.66.

We use the code shown in Listing 3 to write the HTTP request. For demonstration purposes, we use a more-or-less hardwired HTTP request found in the REQUEST_TEMPLATE variable. Because the machine to which we're connecting may change, we do need to fill in the Host header. This is done using snprintf(). We then use SSL_write() to send the data to the server. SSL_write()'s API is more or less the same as write(), except that we pass in the SSL object instead of the file descriptor.

Listing 3. Writing the HTTP Request
(/files/linuxjournal.com/linuxjournal/articles/048/4822/4822l3.html)

Experienced TCP programmers will notice that instead of looping around the write, we throw an error if the return value doesn't equal the value we're trying to write. In blocking mode, SSL_write() semantics are all or nothing; the call won't return until all the data is written or an error occurs, whereas write() may only write part of the data. The SSL_MODE_ENABLE_PARTIAL_WRITE flag (not used here) enables partial writes, in which case you'd need to loop.

In old-style HTTP/1.0, the server transmits its response and then closes the connection. In later versions, persistent connections that allow multiple sequential transactions on the same connection were introduced. For convenience and simplicity we will not use persistent connections. We omit the header that allows them, causing the server to use a connection close to signal the end of the response. Operationally, this means that we can keep reading until we get an end of file, which simplifies matters considerably.

OpenSSL uses the SSL_read() API call to read data, as shown in Listing 4. As with read(), we simply choose an appropriate-sized buffer and pass it to SSL_read(). Note that the buffer size isn't really that important here. The semantics of SSL_read(), like the semantics of read(), are that it returns the data available, even if it's less than the requested amount. On the other hand, if no data is available, then the call to read blocks.

Listing 4. Reading the Response
(/files/linuxjournal.com/linuxjournal/articles/048/4822/4822l4.html)

The choice of BUFSIZZ, then, is basically a performance trade-off. The trade-off is quite different here from when we're simply reading from normal sockets. In that case, each call to read() requires a context switch into the kernel. Because context switches are expensive, programmers try to use large buffers to reduce them. However, when we're using SSL the number of calls to read(), and hence context switches, is largely determined by the number of records the data was written in rather than the number of calls to SSL_read().

For instance, if the client wrote a 1000-byte record and we call SSL_read() in chunks of 1 byte, then the first call to SSL_read() will result in the record being read in, and the rest of the calls will just read it out of the SSL buffer. Thus, the choice of buffer size is less significant when we're using SSL than with normal sockets. If the data were written in a series of small records, you might want to read all of them at once with a single call to read(). OpenSSL provides a flag, SSL_CTRL_SET_READ_AHEAD, that turns on this behavior.

Note the use of the switch on the return value of SSL_get_error(). The convention with normal sockets is that any negative number (typically -1) indicates failure, and that one then checks errno to determine what actually happened. Obviously errno won't work here because that only shows system errors, and we'd like to be able to act on SSL errors. Also, errno requires careful programming in order to be threadsafe.

Instead of errno, OpenSSL provides the SSL_get_error() call. This call lets us examine the return value and figure out whether an error occurred and what it was. If the return value was positive, we've read some data, and we simply write it to the screen. A real client would parse the HTTP response and either display the data (e.g., a web page) or save it to disk. However, none of this is interesting as far as OpenSSL is concerned, so we won't show any of it here.

If the return value was zero, this does not mean that there was no data available. In that case, we would have blocked, as discussed above. Rather, it means that the socket is closed, and there never will be any data available to read. Thus, we exit the loop.

If the return value was something negative, then some kind of error occurred. There are two kinds of errors we're concerned with: ordinary errors and premature closes.

We use the SSL_get_error() call to determine which kind of error we have. Error handling in our client is pretty primitive, so with most errors we simply call berr_exit() to print an error message and exit. Premature closes have to be handled specially.

TCP uses a FIN segment to indicate that the sender has sent all of its data. SSL version 2 simply allowed either side to send a TCP FIN to terminate the SSL connection. This allowed for a truncation attack; the attacker could make it appear that a message was shorter than it was simply by forging a TCP FIN. Unless the victim had some other way of knowing what message length to expect, he or she would simply believe the length was correct.

In order to prevent this security problem, SSLv3 introduced a “close_notify” alert. The close_notify is an SSL message (and therefore secured) but is not part of the data stream itself and so is not seen by the application. No data may be transmitted after the close_notify is sent.

Thus, when SSL_read() returns 0 to indicate that the socket has been closed, this really means that the close_notify has been received. If the client receives a FIN before receiving a close_notify, SSL_read() will return with an error. This condition is called a premature close.

A naïve client might decide to report an error and exit whenever it received a premature close. This is the behavior that is implied by the SSLv3 specification. Unfortunately, sending premature closes is a rather common error, particularly common with clients. Thus, unless you want to be reporting errors all the time, you often have to ignore premature closes. Our code splits the difference. It reports the premature close on stderr but doesn't exit with an error.

If we read the response without any errors, then we need to send our own close_notify to the server. This is done using the SSL_shutdown() API call. We'll cover SSL_shutdown() more completely when we talk about the server, but the general idea is simple: it returns 1 for a complete shutdown, 0 for an incomplete shutdown and -1 for an error. Since we've already received the server's close_notify, about the only thing that can go wrong is that we have trouble sending our close_notify. Otherwise SSL_shutdown() will succeed (returning 1).

Finally, we need to destroy the various objects we've allocated. Since this program is about to exit, thus freeing the objects, this isn't strictly necessary, but it would be in a more general program.

Comments

how can check certificate against CRL in Linux using SSL command (/article/4822#comment-339418)

Submitted by laxman (not verified) on Jul 09, 2009.

Hi,

How can client checks certificate against CRL(certification revocation List) using SSL libray functions.

I have loaded certifcate using X509_load_cert_crl_file(pLookup,"crl.pem",X509_FILETYPE_PEM);

but i am not getting how client can checks....

Thanks & Regards,
Laxman

[Great article, need some info](/article/4822#comment-320560) (/article/4822#comment-320560)
Submitted by saleem (not verified) on Apr 03, 2008.

Hi,
Your article is simple and yet very informative.
Thanks for the article.

I am new to openssl, i am stuck with a problem. I need some info.

Due to some reasons the cert info is available in a buffer rather than a file.
My client needs to load this and establish communication with the server using openssl.

The buffer looks something like this....
```
char *gacacert =
"MIICLzCCAiswggGUoAMCAQICBgEYgSDT3DANBgkqhkiG9w0BAQUFADA0MRAwDgYD\n\
VQQKEwdlbnRydXN0MQwwCgYDVQQLEwNlbmcxEjAQBgNVBAMTCWdhTG9jYWxDQTAe\n\
Fw0wODAzMDUyMjQ3MzVaFw0yODAyMjkyMjQ3MzVaMDQxEDAOBgNVBAoTB2VudHJ1\n\
c3QxDDAKBgNVBAsTA2VuZzESMBAGA1UEAxMJZ2FMb2NhbENBMIGfMA0GCSqGSIb3\n\
DQEBAQUAA4GNADCBiQKBgQDW4ONrqPZ/Hc9Ft/vL1eD76XpbxhdmAezpjGK0aWa2\n\
2QCkDD6IpU3VxpW93+i8em2zgCV5fujbcJuNebk+Y24q3w8FVbba7BZGcaoatB99\n\
vdZ0gp/t/DXq9KsdxdlE2W/mKBCvxkkMsEnm5kHeHZXByouqPvIXGBsJORCH2ahB\n\
vwIDAQABo0gwRjASBgNVHRMBAf8ECDAGAQH/AgEAMBEGCWCGSAGG+EIBAQQEAwIA\n\
JDAdBgNVHQ4EFgQUIZVCc+92iSwt3CD3P9TYIJB6pLQwDQYJKoZIhvcNAQEFBQAD\n\
gYEAjZq3mZ/Q6F26BBd74Q5lJcABGTM4nB1mThaCJk//dLx6WhmWoXJoZD0//nYM\n\
UDvISCc4KtMZoe5qkO/BKJs9IwsXQyZiPl5bAtcfN6OmSe+fmNPMUKD1ck8l7WLu\n\
7k6hlBwrIIi05KhiYLY5i4ZbVh0+DyjIkXbv2GJj+g0CrEE=";
```

Could some one tell me how to load this buffer, communicate with the server and perform the handshake?

Any code sample/example will be great.

Thanks,
Saleem

[@ saleem Hi I am facing](/article/4822#comment-344862) (/article/4822#comment-344862)
Submitted by Rakesh (not verified) on Nov 02, 2009.

@ saleem

Hi

I am facing with the same issue.

I tried using this function to create a mem BIO which can be used to read from the memory.
BIO_new_mem_buf((char *){YOUR STRING},-1);

I was able to read the string from the certificate. But the problem was that when i pass if to SSL_CTX_use_certificate_chain_file it fails.

Please send me any info if you have found a solution already

Thanks
Rakesh

[SSLcontext with BSD Sockets](/article/4822#comment-252868) (/article/4822#comment-252868)
Submitted by Xtremejames183 (not verified) on Jun 05, 2007.

Hi all,
Great article but i want to know how to attach SSL Context to BSD sockets(socket(),bind(),listen(),connect()..)
Thanks for help

[Nice work, Eric. Thank](/article/4822#comment-252588) (/article/4822#comment-252588)
Submitted by Anonymous (not verified) on Jun 04, 2007.

Nice work, Eric. Thank you.

Just a comment: You looks like John Petrucci (guitarist of Dream Theater band)

Thanks for your nice (/article/4822#comment-193185)
Submitted by duanjigang (not verified) on Nov 01, 2006.

Thanks for your nice artical, i am learning it

freeing of returned resource in check_cert() (/article/4822#comment-154868)
Submitted by manaz (not verified) on Aug 04, 2006.

nice article, but there is a little memory leak in the check_cert() function. pointer returned from SSL_get_peer_certificate must be freed according to man page:

The reference count of the X509 object is incremented by one, so that it will not be destroyed when the session containing the peer certificate is freed. The X509 object must be explicitly freed using X509_free().

Re: An Introduction to OpenSSL Programming, Part I of II (/article/4822#comment-937)
Submitted by Anonymous on Mar 24, 2004.

Nice article

need more info (/article/4822#comment-30306)
Submitted by Student (not verified) on Apr 15, 2005.

can u please send me the link of where the next part is ?? that is part 2 of 2 ... thanks for the wonderful tutorial

Re: An Introduction to OpenSSL Programming, Part I of II (/article/4822#comment-938)
Submitted by Anonymous on Oct 18, 2004.

huh!! nice..........

very good , makes it look all (/article/4822#comment-121690)
Submitted by Anonymous (not verified) on Jul 28, 2005.

very good , makes it look all so easy! now to put it into practice

Post new comment

Subject:

Comment: *

Allowed HTML tags: <a> <em> <strong> <cite> <code> <pre> <ul> <ol> <li> <dl> <dt> <dd> <i> <b>
Lines and paragraphs break automatically.
Use to create page breaks.

More information about formatting options (/filter/tips)

Preview