

# An Introduction to OpenSSL Programming (Part I)

Eric Rescorla

RTFM, Inc.

ekr@rtfm.com

Version 1.0: October 5, 2001

## 1 Introduction

The quickest and easiest way to secure a TCP-based network application is with SSL. If you're working in C, your best choice is probably to use OpenSSL, (the web site is at <http://www.openssl.org/>). OpenSSL is a free (BSD-style license) implementation of SSL/TLS based on Eric Young's SSLeay package. Unfortunately, the documentation and sample code distributed with OpenSSL leaves something to be desired. Where they exist, the manual pages are pretty good, but they often miss the big picture, as manual pages are intended as a reference, not a tutorial.

We provide an introduction to OpenSSL programming. The OpenSSL API is vast and complicated so we don't attempt to provide anything like complete coverage. Rather, the idea is to teach you enough to work effectively from the manual pages. In this article, the first of two, we build a simple Web client and server pair that demonstrates the basic features of OpenSSL. In the second article we introduce a number of advanced features, such as session resumption and client authentication.

We assume that you're already familiar with SSL and HTTP at least at a conceptual level. If you're not, a good place to start is with the RFCs (see the end of this article for references).

## 2 Source Code

For space reasons, this article only includes excerpts from the source code. The complete source code is available in machine-readable format from the author's web site at <http://www.rtfm.com/openssl-examples/>

## 3 Our Programs

Our client is a simple HTTPS (see RFC 2818) client. It initiates an SSL connection to the server and then transmits an HTTP request over that connection. It then waits for the response from the server and prints it to the screen. This is a vastly simplified version of the functionality found in programs like fetch and cURL.

The server program is a simple HTTPS server. It waits for TCP connections from clients. When it accepts one it negotiates an SSL connection. Once the connection is negotiated, it reads the client's HTTP request. It then transmits the HTTP response to the client. Once the response is transmitted it closes the connection.

### 3.1 Context Initialization

Our first task is to set up a context object (an `SSL_CTX`). This context object is then used to create a new connection object for each new SSL connection. It is these connection objects which are used to do SSL handshakes, reads, and writes.

This approach has two advantages. First, the context object allows many structures to be initialized only once, improving performance. In most applications, every SSL connection will use

the same keying material, certificate authority (CA) list, etc. Rather than reloading this material for every connection, we simply load it into the context object at program startup. When we wish to create a new connection, we can simply point that connection to the context object. The second advantage of having a single context object is that it allows multiple SSL connections to share data, such as the SSL session cache used for session resumption.

Context initialization consists of four primary tasks, all performed by the `initialize_ctx()` function, shown in Figure 1.

---

```
42 SSL_CTX *initialize_ctx(keyfile,password)
43     char *keyfile;
44     char *password;
45     {
46         SSL_METHOD *meth;
47         SSL_CTX *ctx;
48
49         if(!bio_err){
50             /* Global system initialization*/
51             SSL_library_init();
52             SSL_load_error_strings();
53
54             /* An error write context */
55             bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
56         }
57
58         /* Set up a SIGPIPE handler */
59         signal(SIGPIPE,sigpipe_handle);
60
61         /* Create our context*/
62         meth=SSLv23_method();
63         ctx=SSL_CTX_new(meth);
64
65         /* Load our keys and certificates*/
66         if(!(SSL_CTX_use_certificate_chain_file(ctx,
67             keyfile)))
68             berr_exit("Can't read certificate file");
69
70         pass=password;
71         SSL_CTX_set_default_passwd_cb(ctx,
72             password_cb);
73         if(!(SSL_CTX_use_PrivateKey_file(ctx,
74             keyfile,SSL_FILETYPE_PEM)))
75             berr_exit("Can't read key file");
76
77         /* Load the CAs we trust*/
78         if(!(SSL_CTX_load_verify_locations(ctx,
79             CA_LIST,0)))
80             berr_exit("Ca't read CA list");
81 #if (OPENSSL_VERSION_NUMBER < 0x0090600fL)
82     SSL_CTX_set_verify_depth(ctx,1);
83 #endif
84
85     return ctx;
86     }
```

---

**Figure 1** initialize\_ctx()

## Initialize the library and create the context

Before OpenSSL can be used for anything, the library as a whole must be initialized. This is accomplished with `SSL_library_init()`, which primarily loads up the algorithms that OpenSSL will be using. If we want good reporting of errors, we also need to load the error strings using `SSL_load_error_strings()`. Otherwise, we won't be able to map OpenSSL errors to strings.

We also create an object to be used as an error printing context. OpenSSL uses an abstraction called a BIO object for input and output. This allows the programmer to use the same functions for different kinds of I/O channels (sockets, terminal, memory buffers, etc.) merely by using different kinds of BIO objects. In this case we create a BIO object attached to `stderr` to be used for printing errors.

## Load our own keys

If you're writing server--or a client which is able to perform client authentication--you'll need to load your own public/private key pair and the associated certificate. The certificate is stored in the clear and is loaded together with the CA certificates forming the certificate chain using `SSL_CTX_use_certificate_chain_file()`. We use `SSL_CTX_use_PrivateKey_file()` to load the private key. For security reasons, the private key is usually encrypted under a password. If the key is encrypted, the password callback (set using `SSL_CTX_set_default_passwd_cb()`) will be called to obtain the password.

## Load root CA list

If you're going to be authenticating the host you're connected to OpenSSL needs to know what certificate authorities (CAs) you trust. The `SSL_CTX_load_verify_locations()` call is used to load the CAs.

## Load Randomness

In order to have good security, SSL needs a good source of strong random numbers. In general, it's the responsibility of the application to supply some seed material for the random number generator. However, OpenSSL automatically uses `/dev/urandom` to seed the RNG, if it is available. Since `/dev/urandom` is standard on Linux, we don't have to do anything for this--which is convenient since gathering random numbers is tricky and easy to screw up. Note that if you're on a system other than Linux you may get an error at some point because the random number generator is unseeded. OpenSSL's `rand(3)` manual page provides more information.

## 4 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 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.

## Handshake

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 in Figure 2.

---

```
114     /* Connect the TCP socket*/
115     sock=tcp_connect(host,port);
116
117     /* Connect the SSL socket */
118     ssl=SSL_new(ctx);
119     sbio=BIO_new_socket(sock,BIO_NOCLOSE);
120     SSL_set_bio(ssl,sbio,sbio);
121     if(SSL_connect(ssl)<=0)
122         berr_exit("SSL connect error");
123     if(require_server_auth)
124         check_cert(ssl,host);
```

---

**Figure 2** connecting to the server

## Server Authentication

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 some of the checks 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 Figure 3.

---

```
31 void check_cert(ssl,host)
32     SSL *ssl;
33     char *host;
34     {
35         X509 *peer;
36         char peer_CN[256];
37
38         if(SSL_get_verify_result(ssl)!=X509_V_OK)
39             berr_exit("Certificate doesn't verify");
40
41         /*Check the cert chain. The chain length
42            is automatically checked by OpenSSL when
43            we set the verify depth in the ctx */
44
45         /*Check the common name*/
46         peer=SSL_get_peer_certificate(ssl);
47         X509_NAME_get_text_by_NID
48             (X509_get_subject_name(peer),
49             NID_commonName, peer_CN, 256);
50         if(strcasecmp(peer_CN,host))
51             err_exit
52                 ("Common name doesn't match host name");
53     }
```

---

**Figure 3** `check_cert()` function

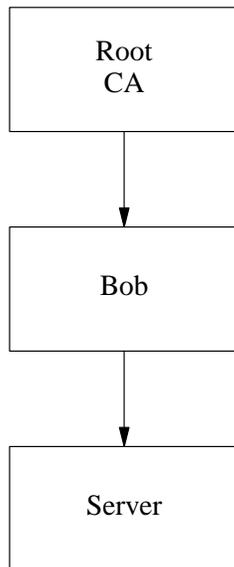
## Server Identity

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 we 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. to

Since most certificates still contain the domain name in Common Name rather than 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 host name we're connecting to. If they don't match, something is wrong and we exit.

## Chain Length

Before version 0.9.5, OpenSSL was subject to a certificate extension attack. To see what this is, consider the case where you a server authenticates with a certificate signed by Bob, as shown in Figure 4. Bob isn't one of your CAs but his certificate is signed by a CA you trust.



**Figure 4** An extended certificate chain

If you accept this certificate you're likely 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 (who you've never heard of) 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're looking at was signed by the CA. 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 `#ifdef-ed` 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 since longer (but properly constructed) chains are becoming more popular.

## When To Check Certificates

The code shown here allows the entire handshake to complete, whether or not the peer has a valid certificate. It's important to note that this is not the only way to check the server's certificate. Only after the handshake completes do we check the certificate and possibly terminate the connection. This approach has the advantage that it's easy to implement but the disadvantage that if we don't like the server's certificate the server just sees a generic error rather than one indicating that the certificate was bad.

An alternative is to use the `SSL_CTX_set_verify()` API call to have OpenSSL require valid certificates during the handshake—if the server presents an invalid certificate the handshake will fail and the server will receive an error indicating that it had a bad certificate. You can also use `SSL_CTX_set_verify()` to set a callback to be called during certificate checking, allowing you to apply extra checks or permit certificates which OpenSSL might otherwise not accept.

## Request

We use the code shown in Figure 5 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 we're connecting to may change we do need to fill in the `Host` header. This is done using the `sprintf()` in line 30. We then use `SSL_write()` to send the data to the server. `SSL_write()` as `write()`, except that we pass in the `SSL` object instead of the file descriptor.

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.

Note: The `SSL_MODE_ENABLE_PARTIAL_WRITE` flag (not used here) enables partial writes, in which case you'd need to loop.

## Response

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 Figure 6. 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.

The choice of `BUFSIZ`, then, is basically a performance tradeoff. The tradeoff is quite different here than 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()`.

---

```

26  request_len=strlen(REQUEST_TEMPLATE)+
27      strlen(host)+6;
28  if(!(request=(char *)malloc(request_len)))
29      err_exit("Couldn't allocate request");
30  sprintf(request,REQUEST_TEMPLATE,
31      host,port);
32
33  /* Find the exact request_len */
34  request_len=strlen(request);
35
36  r=SSL_write(ssl,request,request_len);
37  switch(SSL_get_error(ssl,r)){
38      case SSL_ERROR_NONE:
39      if(request_len!=r)
40          err_exit("Incomplete write!");
41          break;
42      default:
43          berr_exit("SSL write problem");
44  }

```

---

**Figure 5** writing the HTTP request

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.

Note that 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.

---

```

48  while(1){
49      r=SSL_read(ssl,buf,BUFSIZZ);
50      switch(SSL_get_error(ssl,r)){
51          case SSL_ERROR_NONE:
52          len=r;
53          break;
54          case SSL_ERROR_ZERO_RETURN:
55          goto shutdown;
56          case SSL_ERROR_SYSCALL:
57          fprintf(stderr,
58              "SSL Error: Premature close0);
59          goto done;
60          default:
61          berr_exit("SSL read problem");
62      }
63
64      fwrite(buf,1,len,stdout);
65  }

```

---

**Figure 6** reading the response

Note the use of the `switch` on the return value of `SSL_get_error()` in line 12. 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 since 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 thread safe.

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 of course 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 don't show any of it.

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 will never be any data available to read. Thus, we exit the loop.

## Error Handling

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.

## Closure

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 it would simply believe that it had received a shorter message.

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 is called a "premature close".

A naive 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.

## Shutdown

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).

## Cleanup

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

## 5 Server

Our web server is mainly a mirror of the client, but with a few twists. First, we `fork()` in order to let the server handle multiple clients. Second, we use OpenSSL's BIO APIs to read the client's request one line at a time, as well as to do buffered writes to the client. Finally, the server closure sequence is more complicated.

### Accept and Fork

On Linux, the simplest way to write a server that can handle multiple clients is to create a new server process for each client that connects. We do that by calling `fork()` after `accept()` returns. Each new process executes independently and just exits when it's finished serving the client. Although this approach can be quite slow on busy web servers it's perfectly acceptable here. The main server accept loop is shown in Figure 7.

---

```
94     while(1){
95         if((s=accept(sock,0,0))<0)
96             err_exit("Problem accepting");
97
98         if((pid=fork())){
99             close(s);
100        }
101        else {
102            sbio=BIO_new_socket(s,BIO_NOCLOSE);
103            ssl=SSL_new(ctx);
104            SSL_set_bio(ssl,sbio,sbio);
105
106            if((r=SSL_accept(ssl)<=0))
107                berr_exit("SSL accept error");
108
109            http_serve(ssl,s);
110            exit(0);
111        }
112    }
```

---

**Figure 7** server accept loop

### Server Accept

After forking and creating the SSL object, the server calls `SSL_accept()` which causes OpenSSL to perform the server side of the SSL handshake. As with `SSL_connect()`, because we're using blocking sockets, `SSL_accept()` will block until the entire handshake has completed. Thus, the only situation in which `SSL_accept()` will return is when the handshake has completed or an error has been detected. `SSL_accept()` returns 1 for success and 0 or negative for error.

### Buffered I/O

OpenSSL's BIO objects are to some extent stackable. Thus, we can wrap an SSL object in a BIO (the `ssl_bio` object) and then wrap that BIO in a buffered BIO object, as shown in Figure 8. This allows us to perform buffered reads and writes on the SSL connection by using the `BIO_*` functions on the new io object. At this point you might ask: why is this good? Primarily, it's a matter of programming convenience: It lets the programmer work in natural units (lines and characters) rather than SSL records.

---

```
13     io=BIO_new(BIO_f_buffer());
14     ssl_bio=BIO_new(BIO_f_ssl());
15     BIO_set_ssl(ssl_bio,ssl,BIO_CLOSE);
16     BIO_push(io,ssl_bio);
```

---

**Figure 8** creating a buffered BIO

## Request

An HTTP request consists of a request line followed by a bunch of header lines and an optional body. The end of the header lines is indicated by a blank line (i.e., a pair of CRLFs, though sometimes broken clients will send a pair of LFs instead). The most convenient way to read the request line and headers is to read one line at a time until you see a blank line. We can do this using the OpenSSL `BIO_gets()` call, as shown in Figure 9.

The `BIO_gets()` call behaves analogously to the `stdio fgets()` call. It takes an arbitrary size buffer and a length and reads a line from the SSL connection into that buffer. The result is always null terminated (but includes the terminating LF). Thus, we simply read one line at a time until we get a line which consists of simply a LF or a CRLF.

Because we use a fixed length buffer, it is possible though unlikely that we will get an over-long line. In that event the long line will be split over two lines. In the (extremely unlikely) event that the split happens right before the CRLF, the next line we read will consist only of the CRLF from the previous line. In this case we'll be fooled into thinking that the headers have finished prematurely. A real Web server would check for this case but it's not worth doing here. Note that no matter what the incoming line length there's no chance of a buffer overrun. All that can happen is that we'll misparse the headers.

Note that we don't really DO anything with the HTTP request. We just read it and discard it. A real implementation would read the request line and the headers, figure out if there was a body and read that too. However, none of these things show anything interesting about SSL so they don't add anything to this demonstration.

---

```
18     while(1){
19         r=BIO_gets(io,buf,BUFSIZZ-1);
20
21         switch(SSL_get_error(ssl,r)){
22             case SSL_ERROR_NONE:
23                 len=r;
24                 break;
25             default:
26                 berr_exit("SSL read problem");
27         }
28
29         /* Look for the blank line that signals
30            the end of the HTTP headers */
31         if(!strcmp(buf,"
32             !strcmp(buf,"0"))
33             break;
34     }
```

---

**Figure 9** reading the request

## Response

The next step is to write the HTTP response and close the connection. Figure 10 shows this code. Note that we're using `BIO_puts()` instead of `SSL_write()`. This allows us to write

the response one line at a time but have the entire response written as a single SSL record. This is important because the cost of preparing an SSL record for transmission (computing the integrity check and encrypting it) is quite significant. Thus, it's a good idea to make the records as large as possible.

It's worth noting a couple of subtleties about using this kind of buffered write:

1. You need to flush the buffer before you close. The SSL object has no knowledge that you've layered a BIO on top of it, so if you destroy the SSL connection you'll just leave the last chunk of data sitting in the buffer. The `BIO_flush()` call in line 46 takes care of this.
2. By default, OpenSSL uses a 1024-byte buffer size for buffered BIOs. Because SSL records can be up to 16K bytes long, using a 1024-byte buffer cause excessive fragmentation (and hence lower performance.) You can use the `BIO_ctrl()` API to increase the buffer size.

---

```
36     if((r=BIO_puts
37         (io,"HTTP/1.0 200 OK\r\n"))<0)
38         err_exit("Write error");
39     if((r=BIO_puts
40         (io,"Server: EKRServer\r\n\r\n"))<0)
41         err_exit("Write error");
42     if((r=BIO_puts
43         (io,"Server test page\r\n"))<0)
44         err_exit("Write error");
45
46     if((r=BIO_flush(io))<0)
47         err_exit("Error flushing BIO");
```

---

**Figure 10** writing the response

## Shutdown

Once we've finished transmitting the response we need to send our `close_notify`. As before, this is done using `SSL_shutdown()`. Unfortunately, things get a bit trickier when the server closes first. Our first call to `SSL_shutdown()` sends the `close_notify` but doesn't look for it on the other side. Thus, it returns immediately but with a value of 0, indicating that the closure sequence isn't finished. It's then the application's responsibility to call `SSL_shutdown()` again.

It's possible to have two attitudes here:

1. We've seen all of the HTTP request that we care about. we're not interested in anything else. Hence, we don't care whether the client sends a `close_notify` or not.
2. We strictly obey the protocol and expect others to as well. Thus, we require a `close_notify`.

If we have the first attitude then life is simple: We call `SSL_shutdown()` to send our `close_notify` and then exit right away, regardless of whether the client has sent one or not. If we take the second attitude (which our sample server does), then life is more complicated, because clients often don't behave correctly.

The first problem we face is that client's often don't send `close_notify`s all. In fact, some clients close the connection as soon as they've read the entire HTTP response (some versions of IE do this). When we send our `close_notify`, the other side may send a TCP RST segment, in which case the program will catch a `SIGPIPE`. We install a dummy `SIGPIPE` handler in `initialize_ctx()` to protect against this problem.

The second problem we face is that the client may not send a `close_notify` immediately in response to our `close_notify`. Some versions of Netscape require you to send a TCP FIN first. Thus, we call `shutdown(s,1)` before we call `SSL_shutdown()` the second time. When called with a "how" argument of 1, `shutdown()` sends a FIN but leaves the socket open for reading. The code to do the server shutdown is shown in Figure 11 (see next page).

## 6 What's Missing

In this article, we've only scratched the surface of the issues involved with using OpenSSL. Here's a (non-exhaustive) list of additional issues.

### Better Certificate Checking

A more sophisticated approach to checking server certificates against the server hostname is to use the X.509 `subjectAltName` extension. In order to make this check, you would need to extract this extension from the certificate and then check it against the hostname. Additionally, it would be nice to be able to check host names against wildcarded names in certificates. We don't show how to do this either.

---

```
51     r=SSL_shutdown(ssl);
52     if(!r){
53         /* If we called SSL_shutdown() first then
54            we always get return value of '0'. In
55            this case, try again, but first send a
56            TCP FIN to trigger the other side's
57            close_notify*/
58         shutdown(s,1);
59         r=SSL_shutdown(ssl);
60     }
61
62     switch(r){
63     case 1:
64         break; /* Success */
65     case 0:
66     case -1:
67     default:
68         berr_exit("Shutdown failed");
69     }
```

---

**Figure 11** calling `SSL_shutdown()`

### Better Error Handling

Note that these applications handle errors simply by exiting with an error. A real application would, of course, be able to recognize errors and signal them to the user or some audit log rather than just exiting.

## 6.1 Coming Soon

In the second part, we'll be discussing a number of advanced OpenSSL features, including...

### Session Resumption

SSL handshakes are expensive but SSL provides a feature called session resumption that allows you to bypass the full handshake for a peer you've communicated with before.

### Multiplexed and Non-blocking I/O

We used blocking sockets for all of our reads and writes. This means that while the application is waiting for data from the peer it's completely stalled. This wouldn't be acceptable in a real GUI application like a browser. Instead, you need to use the `select()` call to multiplex between SSL and other sources of I/O.

## Client Authentication

Server authentication occurs as a part of almost every SSL connection. However, SSL also offers certificate-based client authentication, at the server's request.

## 6.2 External Resources

Parts of this article were adapted from my book "SSL and TLS: Designing and Building Secure Systems", Addison-Wesley 2000. "SSL and TLS" offers a comprehensive guide to the SSL protocol and its applications. See <http://www.rtfm.com/sslbook> for more information.

Machine readable source code for the programs presented here can be downloaded from the author's web site at: <http://www.rtfm.com/openssl-examples/>. The source code is available under a BSD-style license.

You can get OpenSSL from <http://www.openssl.org/>. The docs are online here as well.

The relevant specifications are at:

SSLv2: [http://www.netscape.com/eng/security/SSL\\_2.html](http://www.netscape.com/eng/security/SSL_2.html)

SSLv3: <http://home.netscape.com/eng/ssl3/index.html>

TLS (RFC 2246): <http://www.ietf.org/rfc/rfc2246.txt>

HTTPS (RFC 2818): <http://www.ietf.org/rfc/rfc2818.txt>

## 6.3 Acknowledgements

Thanks to Lisa Dusseault, Steve Henson, Lutz Jaenicke, and Ben Laurie for help with OpenSSL and review of this article.