An Introduction to OpenSSL Programming, Part II 2007-07-26 09:57:46

分类： 网络与安全

By **Eric Rescorla** on Tue, 2001-10-09 01:00. **Software**

Part two of the series that started in the September 2001 issue.

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 (**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 leave 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. Here, we provide an introduction to OpenSSL programming. The OpenSSL API is vast and complicated, so we don't attempt to provide complete coverage. Rather, the idea is to teach you enough to work effectively from the manual pages. In the first part, published in the September issue of Linux Journal, we introduced the basic features of OpenSSL. In this article we show how to use a number of advanced features such as session resumption and client authentication.

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 **www.rtfm.com/openssl-examples**.

Our Programs

For most of this article we'll be extending the simple client/server pair (wclient and wserver) we presented in Part 1 to create two new programs: wclient2 and wserver2. Like wclient, wclient2 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.

wserver2 is a simple HTTPS server: it waits for TCP connections from clients, and 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.

Towards the end of this article we'll show a more interactive client (sclient) that is usable for debugging or simple remote login.

Session Resumption

When a client and server establish an SSL connection for the first time, they need to establish a shared key called the master_secret. The master_secret is then used to create all the bulk encryption keys used to protect the traffic. The master_secret is almost invariably established using one of two public key algorithms: RSA or Diffie-Hellman (DH). Unfortunately, both of these algorithms are quite slow--on my Pentium II/400 a single RSA operation takes 19 ms. DH can be even slower.

An operation that takes 19 ms may not sound that expensive, but if it has to be done for every connection, it limits the server's throughput to less than 50 connections/second. Without SSL, most web servers can handle hundreds of connections a second. Thus, having to do a key exchange for every client seriously degrades the performance of a web server. In order to improve performance, SSL contains a "session resumption" feature that allows a client/server pair to skip this time consuming step if they have already established a master_secret in a previous connection.

The performance of RSA is highly asymmetric. Operations performed with the private key (such as when the server decrypts the shared key) are much slower than operations performed with the public key. Thus, in most situations most of the computational load is on the server.

What's a Session?

SSL makes a distinction between a connection and a session. A connection represents one specific communications channel (typically mapped to a TCP connection), along with its keys, cipher choices, sequence number state, etc. A session is a virtual construct representing the negotiated algorithms and the master_secret . A new session is created every time a given client and server go through a full key exchange and establish a new master_secret.

Multiple connections can be associated with a given session. Although all connections in a given session share the same master_secret, each has its own encryption keys. This is absolutely necessary for security reasons because reuse of bulk keying material can be extremely dangerous. Resumption allows the generation of a new set of bulk keys and IVs from a common master_secret because the keys depend on the random values, which are fresh for each connection. The new random values are combined with the old master_secret to produce new keys.

How It Works

The first time a client and server interact, they create both a new connection and a new session. If the server is prepared to resume the session, it assigns the session a session_id and transmits the session_id to the client during the handshake. The server caches the master_secret for later reference. When the client initiates a new connection with the server, it provides the session_id to the server. The server can choose to either resume the session or force a full handshake. If the server chooses to resume the session, the rest of the handshake is skipped, and the stored master_secret is used to generate all the cryptographic keys.

Session Resumption on the Client

Listing 1 shows the minimal OpenSSL code required for a client to do session resumption. OpenSSL uses a SESSION object to store the session information, and SSL_get1_session() allows us to get the SESSION for a given SSL object. Once we have obtained the SESSION object we shut down the original connection (using SSL_shutdown()) and create a new SSL object. We use SSL_set_session() to attach the SESSION to the new SSL object before calling SSL_connect(). This causes OpenSSL to attempt to resume the session when it connects to the server. This code is activated by using the -r flag to wclient2.

Listing 1. Reconnect to the Server with Resumed Session

```
144     /* Now hang up and reconnect, if requested */
145     if(reconnect) {
146       sess=SSL_get1_session(ssl); /*Collect the session*/
147       SSL_shutdown(ssl);
148       SSL_free(ssl);
149       close(sock);
150
151       sock=tcp_connect(host,port);
152       ssl=SSL_new(ctx);
153       sbio=BIO_new_socket(sock,BIO_NOCLOSE);
154       SSL_set_bio(ssl,sbio,sbio);
155       SSL_set_session(ssl,sess); /*And resume it*/
156       if(SSL_connect(ssl)<=0)
157         berr_exit("SSL connect error (second connect)");
158       check_cert(ssl,host);
159     }
```

In OpenSSL SESSION objects are reference counted so they can be freed whenever the last reference is destroyed. SSL_get1_session() increments the SESSION reference count, thus allowing SESSION objects to be

used after the SSL is freed.

Of course, this code isn't suitable for a production application because it will only work with a single server. A client can only resume sessions with the same server it created them with, and this code makes no attempt to discriminate between various servers because the server name is constant for any program invocation. In a real application, you would want to have some sort of lookup table that maps hostname/port pairs to SESSION objects.

Session Resumption on the Server

OpenSSL provides a mechanism for automatic session resumption on the server. Each SSL_CTX has a session cache. Whenever a client requests resumption of a given session, the server looks in the cache. As long as two SSL objects share a cache, session resumption will work automatically with no additional intervention from the programmer. In order for the session cache to function correctly, the programmer must associate each SSL object or the SSL_CTX with a session ID context. This context is simply an opaque identifier that gets attached to each session stored in the cache. We'll see how the session ID context works when we discuss client authentication and rehandshake. Listing 2 shows the appropriate code to set the session ID context.

Listing 2. Activating the Server Session Cache

```
149     SSL_CTX_set_session_id_context(ctx,
150       (void*)&s_server_session_id_context,
151       sizeof s_server_session_id_context);
```

Unfortunately, OpenSSL's built-in session caching is inadequate for most production applications, including the ordinary version of wserver. OpenSSL's default session cache implementation stores the sessions in memory. However, we're using a separate process to service each client, so the session data won't be shared and resumption won't work. The -n flag to wserver2 will stop it from forking a new process for each connection. This serves to demonstrate session caching. Without the fork() the server can handle only one client at a time, which makes it much less useful for real-world applications.

We also have to slightly modify the server read loop so that SSL_ERROR_ZERO_RETURN causes the connection to be shut down, rather than an error and an exit. wserver assumes that the client will always send a request, and so exits with an error when the client shuts down the first connection. This change makes wserver2 handle the close properly where wserver does not.

A number of different techniques can be used to share session data between processes. OpenSSL does not provide any support for this sort of session caching, but it does provide hooks for programmers to use their own session caching code. One approach is to have a single session server. The session server stores all of its session data in memory. The SSL servers access the session server via interprocess communication mechanisms, typically some flavor of sockets. The major drawback to this approach is that it requires the creation of some entirely different server program to do this job, as well as the design of the communications protocol used between the SSL server and the session server. It can also be difficult to ensure access control so that only authorized server processes can obtain access.

Another approach is to use shared memory. Many operating systems provide some method of allowing processes to share memory. Once the shared memory segment is allocated, data can be accessed as if it were ordinary memory. Unfortunately, this technique isn't as useful as it sounds because there's no good way to allocate out of the shared memory pool, so the processes need to allocate a single large segment and then statically address inside of it. Worse yet, shared memory access is not easily portable.

The most commonly used approach is to simply store the data on a file on the disk. Then, each server process can open that file and read and write out of it. Standard file locking routines such as flock() can be used to provide synchronization and concurrency control. One might think that storing the data to disk would be dramatically slower than shared memory, but remember that most operating systems have disk caches, and this data would likely be placed in such a cache.

A variant of this approach is to use one of the simple UNIX key-value databases (DBM and friends) rather than a flat file. This allows the programmer to simply create new session records and delete them, without worrying about placing them in the file. If such a library is used, care must be taken to flush the library buffers after each write, because data stored in the buffers has not been written to disk.

Because OpenSSL has no support for cross-process session caches, each OpenSSL-based application needs to solve this problem on its own. ApacheSSL (based on OpenSSL) uses the session server approach. mod_ssl (also based on OpenSSL) can support either the disk-based database approach or a shared memory approach. The code for all of these approaches is too complicated to discuss here, but see Appendix A of SSL and TLS: Designing and Building Secure Systems for a walkthrough of mod_ssl's session caching code. In any case, you probably don't need to solve this problem yourself. Rather, you should be able to borrow the code from mod_ssl or ApacheSSL.

Client Authentication

Most modes of SSL only authenticate the server (and some infrequently used anonymous modes authenticate neither the client nor the server). However, the server can request that the client authenticate using a certificate. OpenSSL provides the SSL_CTX_set_verify() and SSL_set_verify() API calls, which allow you to configure OpenSSL to require client authentication. The only difference between the calls is that SSL_CTX_set_verify() sets the verification mode for all SSL objects derived from a given SSL_CTX--as long as they are created after SSL_CTX_set_verify() is called--whereas SSL_set_verify() only affects the SSL object it is called on.

SSL_CTX_set_verify() takes three arguments: the SSL_CTX to change, the certificate verification mode and a verification callback. The verification callback is called by OpenSSL for each certificate that is verified. This allows fine control over the verification process but is too complicated to discuss here. Check the OpenSSL man pages for more detail.

We're primarily concerned with the verification mode. The mode is an integer consisting of a series of logically or'ed flags. On the server, these flags have the following effect (on the client the effect is somewhat different):

SSL_VERIFY_NONE--don't do certificate-based client authentication

SSL_VERIFY_PEER--attempt to do certificate-based client but don't require it. Note that you must not set SSL_VERIFY_PEER and SSL_VERIFY_NONE together.

SSL_VERIFY_FAIL_IF_NO_PEER_CERT--fail if the client doesn't provide a valid certificate. This flag must be used with SSL_VERIFY_PEER.

SSL_VERIFY_CLIENT_ONCE--if you renegotiate a connection where the client has authenticated, don't requires client authentication (we won't use this flag but it's mentioned for completeness).

wserver2 offers three client authentication options, set in the switch statement shown in Listing 3. The -c switch requests client auth but doesn't require it, using SSL_VERIFY_PEER only. The -C switch requires client auth using SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT. The third option is more complicated. If the -x switch is specified, we allow the client to connect without requesting client authentication. Once the client has sent its request, we then force renegotiation with client authentication. Thus, this branch of the switch does nothing.

Listing 3. Setting Client Authentication Mode

```
158     switch(client_auth){
159       case CLIENT_AUTH_REQUEST:
160         SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,0);
161         break;
162       case CLIENT_AUTH_REQUIRE:
163         SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER |
164           SSL_VERIFY_FAIL_IF_NO_PEER_CERT,0);
```

```
165         break;
166       case CLIENT_AUTH_REHANDSHAKE:
167         /* Do nothing */
168         break;
169     }
```

At this point you might ask, "Why not just require client authentication from the beginning?". For some applications doing so would be just as good but for a real web server it might not be. Imagine that you have a web server that requires SSL for all requests but has a super-secure section for which you want to require certificate-based authentication. Because the SSL handshake occurs before the HTTP request is transmitted, there's no way to tell in advance which part of the web site the client wants to access, and therefore whether client authentication is required.

The workaround is to allow the client to connect without client authentication. The server then reads the client's request and determines whether or not client authentication is required. If it is, it requests a new SSL handshake and requires client authentication. However, since this code is demonstration code, we don't bother to actually examine the client request. We simply force a rehandshake if the -x flag is specified.

Rehandshaking on the Server

The OpenSSL rehandshake API is unfortunately pretty complicated. In order to understand it you first need to understand a little bit about how SSL rehandshake works. Ordinarily the client initiates the SSL handshake by sending a ClientHello message to the server. The client can initiate a rehandshake simply by sending a new ClientHello. If the server wishes to initiate a rehandshake, it sends a HelloRequest message. When it receives a HelloRequest the client may--or may not--initiate a new handshake. Moreover, handshake messages are part of a different stream of data, so data can be flowing while the handshake is happening.

Listing 4 shows the code necessary to implement the rehandshake. The first thing we do is use SSL_set_verify() to tell the server to require the client to perform client authentication. Unfortunately, this setting only controls what OpenSSL does on new handshakes. If a client were to attempt a resumed handshake, this setting would be ignored. To prevent this problem we use SSL_set_session_id_context() to set the resumption context to s_server_auth_session_id_context, instead of s_server_session_id_context.

It's worth taking a minute to understand why changing the session ID context works. Whenever an SSL object stores a session in the session cache, it tags it with the current ID context. Before resuming a session, OpenSSL checks that the current ID context matches the one stored with the cache--but only if SSL_VERIFY_PEER is set. Therefore, changing our ID context ensures that the client can't simply resume the session it just created, because it was created under a different ID context. However, if the client had previously communicated with the server and established a client authenticated session under s_server_auth_session_id_context, then it could resume that session.

Only checking the session ID context if SSL_VERIFY_PEER is set is somewhat counterintuitive; the intent is to ensure that you don't resume sessions in one context that were client authenticated in another context, perhaps with different certificate checking rules. Presumably the idea is that aside from client authentication, one session is rather like another. This assumption is somewhat questionable; one might similarly wish to prevent a session established with one set of cipher preferences from being resumed in an SSL object with a different set of cipher preferences. The only way to accomplish this separation with OpenSSL is to use a totally different SSL_CTX for each cipher suite policy.

Next, we call SSL_negotiate() to move the connection into renegotiate state. Note that this does not cause the connection to be renegotiated; it merely sets the renegotiate flag in the object so that when we call SSL_do_handshake(), the server sends a HelloRequest. Note that the server doesn't do the entire handshake at this point. That requires a separate call to SSL_do_handshake().

Listing 4. Rehandshake with Client Authentication

```
50    /* Now perform renegotiation if requested */
51    if(client_auth==CLIENT_AUTH_REHANDSHAKE){
52      SSL_set_verify(ssl,SSL_VERIFY_PEER |
53        SSL_VERIFY_FAIL_IF_NO_PEER_CERT,0);
54
55      /* Stop the client from just resuming the
56         un-authenticated session */
57      SSL_set_session_id_context(ssl,
58        (void *)&s_server_auth_session_id_context,
59        sizeof(s_server_auth_session_id_context));
60
61      if(SSL_renegotiate(ssl)<=0)
62        berr_exit("SSL renegotiation error");
63     if(SSL_do_handshake(ssl)<=0)
64        berr_exit("SSL renegotiation error");
65      ssl->state=SSL_ST_ACCEPT;
66      if(SSL_do_handshake(ssl)<=0)
67        berr_exit("SSL renegotiation error");
68    }
```

The only remaining piece to explain is explicitly setting the state to SSL_ST_ACCEPT in line 58. Recall we said that the client isn't required to start a rehandshake when it gets the HelloRequest. Moreover, OpenSSL allows the server to keep sending data while the handshake is happening. However, in this case the whole point of the rehandshake is to get the client to authenticate, so we want to finish the handshake before proceeding. Explicitly setting the state forces the server to wait for the client rehandshake. If we don't set the state, the second call to SSL_do_handshake() will return immediately, and the server will send its data before doing the rehandshake. Having to set internal variables of the SSL object is rather ugly, but it's the standard practice in HTTPS servers such as mod_ssl and ApacheSSL, and it has the virtue causing the server to send an unexpected_message alert if the client does not renegotiate as requested.

OpenSSL does offer a SSL_set_accept_state() call, but it isn't useful here because it clears the current crypto state. Since we're already encrypting data, clearing the crypto state causes MAC errors.

For simplicity's sake we've decided to always force a rehandshake when wserver2 is invoked with the -x flag. However, this can create an unnecessary performance load on the server. What if the client has already authenticated? In that case there's no point in authenticating it again. This situation can occur if the client is resuming a previous session in which client authentication occurred. (It cannot happen if this is the first time the client is connecting because client authentication only happens when the server requests it, and we only request it on the rehandshake).

An enhanced version of wserver2 could check to see if the client already provided a certificate and skip the rehandshake if it had. However, great care must be taken when doing this because of the way OpenSSL resumes sessions. Consider the case of a server that has two different sets of certificate verification rules, each associated with a separate ID context. Because OpenSSL only checks the ID context when SSL_VERIFY_PEER is set--which it is not for our initial handshake--the client could resume a session associated with either context. Thus, in addition to getting the certificate we would need to check that the session being resumed came from the right session ID context. If it didn't, we'd still have to rehandshake to be sure that we get the right kind of client authentication.

Rehandshaking on the Client

Automatic rehandshake support is built into SSL_read(), so the client doesn't have to do anything explicit to

enable rehandshake. However, this doesn't necessarily mean that arbitrary clients will work. In particular, the client must be prepared to handle the SSL_ERROR_WANT_READ return value from SSL_read(). Ordinarily, if the client is using blocking I/O--as we are in wclient2--SSL_read() would never return this error. However, it's required if rehandshaking occurs.

To see why this is, imagine that your client uses select() to determine when there is data available from the server. When select() returns, you call SSL_read(). If the server is requesting rehandshake, however, the data will be a HelloRequest message. SSL_read() consumes the message and performs the handshake. There's no guarantee that there will be any more data available on the socket, so SSL_read() needs to return without any data--hence the SSL_ERROR_WANT_READ return value.

Our original wclient program, despite functioning properly in every other respect, broke utterly when dealing with rehandshake, thereby proving once again that if you haven't tested some feature, it almost certainly doesn't work. The fix in wclient2 is to change the read loop that is shown in Listing 5.

Listing 5. New Client Read Loop with Rehandshake Support

```
48     /* Now read the server's response, assuming
49        that it's terminated by a close */
50     while(1){
51       r=SSL_read(ssl,buf,BUFSIZZ);
52       switch(SSL_get_error(ssl,r)){
53         case SSL_ERROR_NONE:
54           len=r;
55           break;
56         case SSL_ERROR_WANT_READ:
57           continue;
58         case SSL_ERROR_ZERO_RETURN:
59           goto shutdown;
60         case SSL_ERROR_SYSCALL:
61           fprintf(stderr,
62             "SSL Error: Premature close\n");
63           goto done;
64         default:
65           berr_exit("SSL read problem");
66       }
67
68       fwrite(buf,1,len,stdout);
69     }
```

Since this is an HTTP client and it's already written the request, there's no need to wonder if there is data coming from the server. The next traffic on the wire will always be the server's response. Thus, we don't need to select() on the socket. If we get SSL_ERROR_WANT_READ, we just go back to the top of the loop and call SSL_read() again.

Using Client Authentication Information

When using client authentication it's important to understand what it does and doesn't provide. So far, all the server knows is that the client possessed the private key corresponding to some valid certificate. In some cases this might be enough, but in most cases the server wants to know who the client is in order to make authorization decisions.

Checking the client's certificate is roughly similar to the check_cert() function we used to check the server's identity. The server would extract the client's name from the certificate and check it against some access

control list. If the name is on the list, the client will be accorded the appropriate privileges. Otherwise, access will be denied.

Controlling Cipher Suites

SSL offers a multiplicity of cryptographic algorithms. Since they are not equally strong and fast, users often want to choose one algorithm over another. By default, OpenSSL supports a broad variety of ciphers; however, it provides an API for restricting the cipher that it will negotiate. We expose this functionality in wclient2 and wserver2 with the -a flag.

To use the -a flag, the user provides a colon-separated list of ciphers, as in -a RC4-SHA:DEC-CBC3-SHA:DES-CBC- SHA. This string is then passed to OpenSSL via the SSL_CTX_set_cipher_list() function, as shown in Listing 6.

Listing 6. Setting the Cipher List

```
126     /* Set our cipher list */
127     if(ciphers){
128       SSL_CTX_set_cipher_list(ctx,ciphers);
129     }
```

You can get a list of all the algorithms that OpenSSL supports using the openssl command. Try **openssl ciphers**. Also, you might want to try using the -a flag with both client and server. Verify for yourself that if you use the same cipher (or have the same one on both lists) things work and, otherwise, the connection fails.

Multiplexed I/O

Our wclient program is just about the most trivial client program possible because the I/O semantics are so simple. The client always writes everything it has to write, and then reads everything that the server has to read. Reads and writes are never interlaced, and the client just stops and waits for data from the server. This works fine for simple applications, but there are many situations in which it's unacceptable. One such application is a remote login client.

A remote login client such as Telnet or ssh needs to process at least two sources of input: the keyboard and the network. Input from the keyboard and the server can appear asynchronously. That is to say they can appear in any order. This means that the read/write type I/O discipline that we had in wclient is fundamentally inadequate.

It's easy to see this. Consider an I/O discipline analogous to the one we used in wclient, represented by the pseudo-code in Listing 7.

Listing 7. A Broken Client I/O Discipline

```
1    while(1){
2      read(keyboard,buffer);
3      write(server,buffer);
4      read(server,buffer);
5      write(screen,buffer);
6    }
```

Consider the case in which you're remotely logged into some machine and request a directory listing. Your request is a single line (**ls** on UNIX boxes) but the response is a large number of lines. In general, these lines will be written in more than one write. Thus, it may very well take more than one read in order to read them from the server. However, if we use the I/O discipline in Listing 7, we'll run into a problem.

We read the command from the user and the first chunk of the server's response, but after that we get deadlocked. The client is waiting in line 2 for the user to type something, but the user is waiting for the rest of the directory listing. We're deadlocked. In order to break the deadlock, we need some way to know when either the keyboard or the network is ready to read. We can then service that source of input and keep from deadlocking. Conveniently, Linux provides us with a call that does exactly that-- select(2). select() is the

standard tool for doing multiplexed I/O. It lets you determine whether any of a set of sockets is ready to read or write. If you're not familiar with it already, read the man page or consult Richard Stevens's fine book Advanced Programming in the UNIX Environment (Addison-Wesley 1992).

Unfortunately, although select() is a common UNIX idiom, its use with OpenSSL is far from clean and requires understanding of some subtleties of SSL. In order to demonstrate them, we present a new program, sclient.

sclient is a simple model of an SSLized remote access client. It connects to the server and then transfers anything typed at the keyboard to the server, and anything sent from the server to the screen.

Read

The basic problem we're facing is that SSL is a record-oriented protocol. Thus, even if we want to read only one byte from the SSL connection, we still need to read the entire record containing that byte into memory. Without the entire record in hand, OpenSSL can't check the record MAC, so we can't safely deliver the data to the programmer. Unfortunately, this behavior interacts poorly with select(), as shown in Figure 1.

Figure 1. Read Interaction with SSL

The left-hand side of Figure 1 shows the situation when the machine has received a record but it's still waiting in the network buffers. The arrow represents the read pointer that is set at the beginning of the buffer. The bottom row represents data decoded by OpenSSL but not yet read by the program (the SSL buffer). This buffer is currently empty so we haven't shown a box. If the program calls select() at this point, it will return immediately, indicating that a call to read() will succeed. Now, imagine that the programmer calls SSL_read() requesting one byte. This takes us to the situation at the right side of the figure.

As we said earlier, the OpenSSL has to read the entire record in order to deliver even a single byte to the program. In general, the application does not know the size of records, and so its reads will not match the records. Thus, the box in the upper right-hand corner shows that the read pointer has moved to the end of the record. We've read all the data in the network buffer. When the implementation decrypts and verifies the record, it places the data in the SSL buffer. Then it delivers the one byte that the program asked for in SSL_read(). We show the SSL buffer in the lower right-hand corner. The read pointer points somewhere in the buffer, indicating that some of the data is available for reading but some has already been read.

Consider what happens if the programmer calls select() at this point. **select()** is concerned solely with the contents of the network buffer, and that's empty. Thus, as far as select() is concerned there's no data to read. Depending on the exact arguments it's passed, it will either return and say that there's nothing to read or wait for some more network data to become available. In either case we wouldn't read the data in the SSL buffer. Note that if another record arrived, select() would indicate that the socket was ready to read and, we'd have an opportunity to read more data.

Thus, select() is an unreliable guide to whether there is SSL data ready to read. We need some way to determine the status of the SSL buffer. This can't be provided by the operating system because it has no access to the SSL buffers. It must be provided OpenSSL. OpenSSL provides exactly such a function. The function SSL_pending() tells us whether there is data in the SSL buffer for a given socket. Listing 8 shows SSL_pending() in action.

Listing 8. Reading Data Using SSL_Pending()

```
49      /* Now check if there's data to read */
50      if((FD_ISSET(sock,&readfds) && !write_blocked_on_read) ||
51        (read_blocked_on_write && FD_ISSET(sock,&writefds))){
52        do {
53          read_blocked_on_write=0;
54          read_blocked=0;
55
56          r=SSL_read(ssl,s2c,BUFSIZZ);
```

```
57
58          switch(SSL_get_error(ssl,r)){
59            case SSL_ERROR_NONE:
60              /* Note: this call could block, which blocks the
61                 entire application. It's arguable this is the
62                 right behavior since this is essentially a terminal
63                 client. However, in some other applications you
64                 would have to prevent this condition */
65              fwrite(s2c,1,r,stdout);
66              break;
67            case SSL_ERROR_ZERO_RETURN:
68              /* End of data */
69              if(!shutdown_wait)
70                SSL_shutdown(ssl);
71              goto end;
72              break;
73            case SSL_ERROR_WANT_READ:
74              read_blocked=1;
75              break;
76
77              /* We get a WANT_WRITE if we're
78                 trying to rehandshake and we block on
79                 a write during that rehandshake.
80
81                 We need to wait on the socket to be
82                 writeable but reinitiate the read
83                 when it is */
84            case SSL_ERROR_WANT_WRITE:
85              read_blocked_on_write=1;
86              break;
87            default:
88              berr_exit("SSL read problem");
89          }
90
91          /* We need a check for read_blocked here because
92             SSL_pending() doesn't work properly during the
93             handshake. This check prevents a busy-wait
94             loop around SSL_read() */
95        } while (SSL_pending(ssl) && !read_blocked);
96      }
```

The logic of this code is fairly straightforward. **select()** has been called earlier, setting the variable readfds with the sockets that are ready to read. If the SSL socket is ready to read, we go ahead and try to fill our buffer, unless the variable write_blocked_on_read is set (this variable is used when we're rehandshaking and we'll discuss it later). Once we've read some data, we write it to the console. Then we check with SSL_pending() to see if the record was longer than our buffer. If it was, we loop back and read some more data.

Note that we've added a new branch to our switch statement: a check for SSL_ERROR_WANT_READ. What's going on here is that we've set the socket for nonblocking operation. Recall that we said that if you called

read() when the network buffers were empty, it would simply block (wait) until they weren't. Setting the socket to nonblocking causes it to return immediately, saying that it would have blocked.

To understand why we've done this, consider what happens if an SSL record arrives in two pieces. When the first piece arrives, select() will signal that we're ready to read. However, we need to read the entire record in order to return any data, so this is a false positive. Attempting to read all that data will block, leading to exactly the deadlock we were trying to avoid. Instead, we set the socket to non-blocking and catch the error, which OpenSSL translates to SSL_ERROR_WANT_READ.

It's worth noting that the call to fwrite() that we use to write to the console can block. This will cause the entire application to stall. This is reasonable behavior in a terminal client--if the user isn't looking at the screen we want the server to wait for him--but in other applications we might have to make this file descriptor non-blocking and select() on it as well. This is left as an exercise for the reader.

Write

When we're writing to the network we have to face the same sort of inconsistency that we had when reading. Again, the problem is the all-or-nothing nature of SSL record transmission. For simplicity's sake, let's consider the case where the network buffers are mostly full and the program attempts to perform a modest-sized write, say 1K. This is illustrated in Figure 2.

Figure 2. Write Interaction with SSL

Again, the left-hand side of the figure represents our initial situation. The program has 1K to write in some buffer. The write pointer is set at the beginning of that buffer. The SSL buffers are empty. The network buffer is half-full (the shading indicates the full region). The write pointer is set at the beginning. We've deliberately obscured the distinction between the TCP buffers and the size of the TCP window because it's not relevant here. Suffice to say that the program can safely write 512 bytes without blocking.

Now, the program calls SSL_write() with a 1024-byte block. OpenSSL has no way of knowing how much data it can write safely, so it simply formats the buffer as a single record, thus moving the write pointer in the program buffer to the end of the buffer. We can ignore the slight data expansion from the SSL header and MAC, and simply act as if the data to be written to the network was 1024 bytes.

Now, what happens when OpenSSL calls write()? It successfully writes 512 bytes but gets a would_block error when it attempts to write to the end of the record. As a consequence, the write pointer in the SSL buffer is moved halfway across--indicating that half of the data has been written to the network. The network buffer is shaded to indicate that it's completely full. The network write pointer hasn't moved.

We now need to concern ourselves with two questions: first, how does the toolkit indicate this situation to the application, and, second, how does the programmer arrange it so the SSL buffer gets flushed when space is available in the network buffer? The kernel will automatically flush the network buffer when possible, so we don't need to worry about arranging for that. We can use select() to see when there is more space available in the network buffer, and we should therefore flush the SSL buffer.

Once OpenSSL has received a would_block error from the network, it aborts and propagates that error all the way up to the application. Note that this does not mean it throws away the data in the SSL buffer. This is impossible because part of the record might already have been sent. In order to flush this buffer, we must call SSL_write() again with the same buffer that it called the first time (it's permissible to extend the buffer but the start must be the same.) OpenSSL automatically remembers where the buffer write pointer was and only writes the data after the write pointer. Listing 9 shows this process in action.

Listing 9. Client to Server Writes Using OpenSSL

```
98      /* Check for input on the console*/
99      if(FD_ISSET(fileno(stdin),&readfds)){
100         c2sl=read(fileno(stdin),c2s,BUFSIZZ);
101         if(c2sl==0){
```

```
102           shutdown_wait=1;
103           if(SSL_shutdown(ssl))
104             return;
105         }
106         c2s_offset=0;
107       }
108
109       /* If the socket is writeable... */
110       if((FD_ISSET(sock,&writefds) && c2sl) ||
111         (write_blocked_on_read && FD_ISSET(sock,&readfds))) {
112         write_blocked_on_read=0;
113
114         /* Try to write */
115         r=SSL_write(ssl,c2s+c2s_offset,c2sl);
116
117         switch(SSL_get_error(ssl,r)){
118           /* We wrote something*/
119           case SSL_ERROR_NONE:
120             c2sl-=r;
121             c2s_offset+=r;
122             break;
123
124             /* We would have blocked */
125           case SSL_ERROR_WANT_WRITE:
126             break;
127
128             /* We get a WANT_READ if we're
129                trying to rehandshake and we block on
130                write during the current connection.
131
132                We need to wait on the socket to be readable
133                but reinitiate our write when it is */
134           case SSL_ERROR_WANT_READ:
135             write_blocked_on_read=1;
136             break;
137
138               /* Some other error */
139           default:
140             berr_exit("SSL write problem");
141         }
142       }
```

The first thing we need to do is have some data to write. Thus, we check to see if the console is ready to read, and if so, read whatever's there (up to BUFSIZZ bytes) into the buffer c2s, placing the length in the variable c2sl.

If c2sl is nonzero and the network buffers are (at least partially) empty, then we have data to write to the network. As usual, we call SSL_write() with the buffer c2s. As before, if we manage to write some but not all of the data, we simply increment c2s_offset and decrement c2sl.

The new behavior here is that we check for the error SSL_ERROR_WANT_WRITE. This error indicates that we've got unflushed data in the SSL buffer. As we described above, we need to call SSL_write() again with the same buffer, so we simply leave c2sl and c2s_offset unchanged. Thus, the next time SSL_write() is called it will automatically be with the same data.

OpenSSL actually provides a flag called SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER that allows you to call SSL_write() with a different buffer after a would_block error. However, this merely allows you to allocate a new buffer with the same contents. SSL_write() still seeks to the same write pointer before looking for new data.

The Interaction of Multiplexed I/O with Rehandshake

We saw earlier that rehandshaking could cause SSL_write() to return with SSL_ERROR_WANT_READ. Similarly, SSL_read() can return with SSL_ERROR_WANT_WRITE if rehandshake occurs and the write buffers fill up (this is relatively unusual on modern platforms since the write buffers are generally large enough to accommodate typical SSL handshake records). In either case, the correct procedure is to select for read/write-ability and then reissue the offending call. This leads to some confusing logic since it means that under certain circumstances you respond to a socket being ready to read by calling SSL_write().

Let's take the case of SSL_write() first. If SSL_write() returns SSL_ERROR_WANT_READ we set the state variable write_blocked_on_read. Ordinarily, this would cause us to call SSL_read() but because write blocked on read is set, we skip the read loop and fall through to the write loop where the call to SSL_write() is reissued. Note that since the SSL handshake requires a number of reads and writes, it's quite possible that this call won't complete either, and we'll have to reenter the select() loop.

Now consider the case of SSL_read(). If SSL_read() returns SSL_ERROR_WANT_WRITE we set the state variable read_blocked_on_write and reenter the select() loop waiting for the socket to be writable. When it is we arrange to call SSL_read() again (see the second line of the test right after the call to select(). Note that after we call SSL_read() we fall through the the write section of the loop. However, even though the socket is writeable and so the FD ISSET test passes, SSL_write() will only be called if c2sl is nonzero and there's really data to write.

What's Still Missing?

Taken together, these articles demonstrate most of the essentials of writing SSL clients and servers with OpenSSL. However, OpenSSL offers a number of other powerful features that we don't cover, including:

Cryptographic Toolkit

Underlying OpenSSL's SSL implementation is a crypto toolkit implementing all the major cryptographic primitives, including RSA, DH, DES, 3DES, RC4, AES, SHA and MD5, as well as an ASN.1 encoder. Thus, OpenSSL is useful for people writing other kinds of cryptographic applications, even if they don't involve SSL.

Certificate Authority

OpenSSL contains the basic software required to write a certificate authority (CA). A number of CAs have been written on top of OpenSSL, including the free OpenCA project (see the references section).

S/MIME

OpenSSL now includes an S/MIME implementation, allowing it to be used to write secure mail clients. This is still somewhat of a hard hat area--the S/MIME support isn't quite complete yet and neither is the documentation.

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 **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: **www.rtfm.com/openssl-examples**. The source code is available under a BSD-style license.

You can get OpenSSL from **www.openssl.org**. The docs are on-line here as well.
The SSLv2 and SSLv3 specifications are on-line at **www.netscape.com/eng/security/SSL_2.html** and **home.netscape.com/eng/ssl3/index.html** The specifications for TLS (RFC 2246) and HTTPS (RFC 2818) are available at **www.ietf.org/rfc/rfc2246.txt** and **www.ietf.org/rfc/rfc2818.txt**.
The mod_ssl project lives at **www.modssl.org**. The ApacheSSL project lives at **www.apachessl.org**. The OpenCA project lives at **www.openca.org/**.

Acknowledgements

Many thanks to Lutz Jaenicke and Bodo Moeller for help with OpenSSL and catching a number of problems with the example programs. Thanks to Lisa Dusseault for review of this article.
Eric Rescorla has been working in internet security since 1993.