An Introduction to OpenSSL Programming, Part II of II

Oct 09, 2001 By Eric Rescorla (/user/800894)
in

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

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

```
    SSL_CTX_set_session_id_context(ctx,
      (void*)&s_server_session_id_context,
      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.

Comments

Comment viewing options

Threaded list - expanded | Date - newest first | 50 comments per page | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

Block socket with SSL_MODE_AUTO_RETRY (/article/5487#comment-1300)

Submitted by Anonymous on Oct 04, 2004.

If I use block socket with SSL_MODE_AUTO_RETRY
The flag SSL_MODE_AUTO_RETRY will cause read/write operations to only return after the handshake and successful completion.

Do I have to handle the retrying in SSL_Read and SSL_Write? Isn't it easier to do this way? I know it hurts throughput a little, but is it a big deal?

Post new comment

Subject:

Comment: *