Enhancing SSL Performance

Jens G Jensen
CCLRC Rutherford Appleton Laboratory

5 Feb 2006

Abstract

The most commonly deployed library for handling Secure Sockets Layer (SSL) and Transport Layer Security (TLS) [RFC2246] is OpenSSL [OpenSSL]. This paper presents performance tests of the OpenSSL library handling sockets and certificates, and, in particular, sending HTTP over SSL to Apache. The main result of the paper is an analysis of the effect of caching information at various levels of the SSL connection on the client side, thus providing guidelines for speeding up SSL socket connections. Since OpenSSL and libraries built on OpenSSL (e.g. Globus GT2) are ubiquitous, this work will be of interest to anyone writing Grid and SSL clients.

Often SSL tuning information and options are for the server side, including buying hardware acceleration. SSLSwamp (part of the distcache project [Distcache]) is used to stress test servers. However, the OpenSSL library allows optimisations by caching information also on the client side. In this paper we discuss the possibilites and impact.

In (usually chronological) order, and in order of least specific to most specific level, the levels are approximately as follows:

- **Library**: the Library itself is usually loaded dynamically when the client starts up. The library must be initialised, usually error strings are loaded, the engine, if any, is initialised, and the random source is initialised.
- **Context**: The Context is usually the same for each connection. It contains the location of trusted CAs, and acceptable ciphers. For Grid clients and other clients using client authentication, the private key and certificates are also loaded into the context.
- **BIO**: The BIO is the OpenSSL I/O abstraction. In this case, it refers to a socket communicating via TCP/IP to the server. Obviously, to set up a socket BIO, the client needs the hostname and port of the server.
- **SSL**: Next, the SSL layer is initialised on top of the BIO. It performs the SSL negotiation, validating the certificates of the server and, if used, client. The client must extract the hostname from the server's certificate and compare it to the name to which it connected ([RFC2830] section 3.6).

- **Message**: Finally, if all goes well, the server and client can talk to each other over the secure channel. In our tests, the client sends a simple message to the server via HTTP, and obtains a response.

In addition to these, there is the **Session**. The Session is created when the client connects to the server (i.e., is specific to establishing the SSL layer above), and can be *preserved* between connections. In our experiment, we keep it in memory, but it can be serialised. We thus expect a preserved Session to speed up establishing the SSL layer, but not the other layers. Obviously, the server must be configured to support Session preservations.

1 Theory

RSA and other public key calculations are expensive. The SSL protocol uses the public key calculations to, eventually, agree a symmetric key between the server and the client.

Using GSI credentials only speeds up establishing the Context – without it, the SSL library has to ask the user for a passphrase, which is of course *very* slow. One could imagine an "SSL agent" (similar to ssh agents), but again it will only speed up establishing the context. The Session keeps shared secrets from a previous SSL negotiation between a specific server and the client, so should in theory speed up future SSL negotiations between the client and the same server.

2 Experiment

Apache was used as web server in most of this experiment, running with pretty much out-of-the-box configuration. Apache supports three types of SSL shutdown – what they call *standard*, where the server can close the connection but doesn't wait for the client to close its connection, *unclean* where the server does not send any SSL shutdown but just closes the TCP connection (which violates [RFC2246]), and *accurate*, where the server sends an SSL shutdown and waits for the client to respond. Some older browsers never sent the corresponding close notify response, so the server uses *standard* by default.

All tests are run sequentially, and with small message payloads: we're testing the client connection overhead, not the server or the transfer rate.

The table below shows the costs (in seconds, lower is better) of restarting the client (run), reinitialising the library and the RNG (lib), the context (ctx), the BIO or socket (bio), and SSL connection (ssl), and finally at the message level (msg) using keepalive (because the message happens to be HTTP). Each test is done 1000 times. For example, in the "ctx" row, the Library is initialised only once and the Context is initialised 1000 times, and for each time the Context is initialised, the "higher" layers (bio, ssl) are also set up, once each. So for each row, except the very first where the program is loaded but makes no connections, the effect on the server side is that the server receives exactly 1000 messages. The "message" sent in each case is an HTTP 1.1 `HEAD /` request. Moreover, for all rows except the first (where no connections are made) and the last (where the same connection is kept open for all 1000 requests), the server is doing *exactly* the same amount of work – it is doing exactly 1000 SSL negotiations with the client.

Test	**localhost**[1]	**localhost**[2]	**WAN**[1,3]
1000 run[4]	4.6	-	-
1000 run[5]	*344.8*	-[8]	-
1000 run[6]	197.4	-	-
1000 run[7]	189.2	-	-
1000 lib[5]	*2765*	-	-
1000 lib[6]	174.6	173.83	194.4
1000 lib[7]	170.2	170.9	196.3
1000 ctx	167.0	169.4	187.6
1000 bio	168.0	163.0	177.3
1000 ssl	N/A[9]	N/A	N/A
1000 msg	0.27	0.27	0.4

1. Not using SSL sessions.
2. Client attempts to use SSL sessions.
3. On a LAN with completely different machines, the faster machine running the server. Data in this column cannot be compared directly with the localhost column.
4. Starting the program, loading the library but not initialising it or doing any other SSL.
5. The library initialises the random number generator (RNG) from `/dev/random`. The numbers in this test are meaningless because the system runs out entropy and must wait for external sources to refill the pool. Indeed, the author was mousing more during the first of the tests.
6. The library initialises the RNG from `/dev/urandom` which doesn't block but may return poorer quality random seeds, thus compromising security.
7. The library doesn't initialise the RNG at all. Never do that in a real application!
8. Session is kept in memory in this application, so session reuse is not available if the program is restarted.
9. Apache doesn't allow the socket to be reused for another SSL session; see discussion below.

Reusing the socket for another SSL connection (the "ssl" case above) is less interesting with Apache, because Apache is likely to close the socket after it has shut down SSL (this could not be verified in this experiment because Apaache did not do a clean SSL shutdown, see below). Furthermore, on most networks (and certainly the ones tested above), the overhead in establishing sockets is negligible compared to the establishing the SSL connection, so the result from the "bio" row would be very close to that of the "ssl" row.

To reuse the socket for another SSL connection, it is necessary to set Apache to *accurate* shutdown. Otherwise the client will notice it and refuse to reuse the socket. Unfortunately, a bug in Apache 2.0.54 and 55 means that Apache mod_ssl ignores the `ssl-accurate-shutdown` flag.

We summarise the findings:

1. Since one should use the best random source available (e.g., `/dev/random` on Linux, `/dev/srandom` on OpenBSD), it follows that one should start up the client and initialise the library as few times as possible. Consequently, one should start up the program and let the program handle many connections to the servers (possibly containing credentials for more than one user, ie. containing more than one Context).
2. The overhead of (re)starting the program is insignificant compared to the overhead of reinitialising the library. Thus, one should consider the costs of initialising the library when designing secure applications.
3. Relying on Sessions to speed up connections is not a good idea. Indeed, the program reported that each time it tried to reuse the Session it failed, despite the fact that server was set to use the session cache (both with the default dbm and the shm options).
4. Use keepalives if your protocol supports it. No SSL reuse is necessary because there is only one (or $O(1)$) SSL connections.

3 Future directions

- The software used for testing is of course available for others to use, under a BSD licence [sslperf].
- Find out why the Apache server refuses to reuse sessions. And whether it can shut down SSL properly (bugfix).
- Integrate with Globus GSI libraries.
- Use different public key algorithms (e.g., DSA, elliptic curves).
- Investigate other types of authentication or message confidentiality, c.f., [Message level vs SSL layer].

Acknowledgments

The author wishes to thank Owen Synge at RAL for suggesting this work, and Alastair Duncan from the R-GMA group at RAL for interesting dicussions: non-Java R-GMA clients had the same problems and are now using HTTP keepalive. This document typeset with LaTeX.

4 References

sslperf: http://www.escience.clrc.ac.uk/documents/staff/jens_jensen/sslperf.tar.bz2
Message level vs SSL layer: http://grid.racf.bnl.gov/GUMS/gridServPerf2.html
OpenSSL: http://www.openssl.org/
Distcache: http://distcache.sourceforge.net/
RFC2246: http://www.rfc-editor.org/rfc/rfc2246.txt
RFC2830: http://www.rfc-editor.org/rfc/rfc2830.txt