



Introduction

HttpClient provides full support for HTTP over Secure Sockets Layer (SSL) or IETF Transport Layer Security (TLS) protocols by leveraging the [Java Secure Socket Extension \(JSSE\)](#) . JSSE has been integrated into the Java 2 platform as of version 1.4 and works with HttpClient out of the box. On older Java 2 versions JSSE needs to be manually installed and configured. Installation instructions can be found [here](#) .

Standard SSL in HttpClient

Once you have JSSE correctly installed, secure HTTP communication over SSL should be as simple as plain HTTP communication.

```
HttpClient httpClient = new HttpClient();
GetMethod httpget = new GetMethod("https://www.verisign.com/");
try {
    httpClient.executeMethod(httpget);
    System.out.println(httpget.getStatusLine());
} finally {
    httpget.releaseConnection();
}
```

HTTPS communication via an authenticating proxy server is also no different from plain HTTP communication. All the low-level details of establishing a tunneled SSL connection are handled by HttpClient:

```
HttpClient httpClient = new HttpClient();
httpClient.getHostConfiguration().setProxy("myproxyhost", 8080);
httpClient.getState().setProxyCredentials("my-proxy-realm", "myproxyhost",
new UsernamePasswordCredentials("my-proxy-username", "my-proxy-password"));
GetMethod httpget = new GetMethod("https://www.verisign.com/");
try {
    httpClient.executeMethod(httpget);
    System.out.println(httpget.getStatusLine());
} finally {
    httpget.releaseConnection();
}
```

Customizing SSL in HttpClient

The default behaviour of HttpClient is suitable for most uses, however there are some aspects which you may want to configure. The most common requirements for customizing SSL are:

- Ability to accept self-signed or untrusted SSL certificates. This is highlighted by an `SSLException` with the message *Unrecognized SSL handshake* (or similar) being thrown when a connection attempt is made.
- You want to use a third party SSL library instead of Sun's default implementation.

Implementation of a custom protocol involves the following steps:

1. Provide a custom socket factory that implements [org.apache.commons.httpclient.protocol.SecureProtocolSocketFactory](#) interface. The socket factory is responsible for opening a socket to the target server using either the standard or a third party SSL library and performing any required initialization such as performing the connection handshake. Generally the initialization is performed automatically when the socket is created.
2. Instantiate an object of type [org.apache.commons.httpclient.protocol.Protocol](#). The new instance

would be created with a valid URI protocol scheme (https in this case), the custom socket factory (discussed above) and a default port number (typically 443 for https). For example:

```
Protocol myhttps = new Protocol("https", new MySSLSocketFactory(), 443);
```

The new instance of protocol can then be set as the protocol handler for a HostConfiguration. For example to configure the default host and protocol handler for a HttpClient instance use:

```
HttpClient httpClient = new HttpClient();
httpClient.getHostConfiguration().setHost("www.whatever.com", 443, myhttps);
GetMethod httpget = new GetMethod("/");
try {
    httpClient.executeMethod(httpget);
    System.out.println(httpget.getStatusLine());
} finally {
    httpget.releaseConnection();
}
```

3. Finally, you can register your custom protocol as the default handler for a specific protocol designator (eg: https) by calling the Protocol.registerProtocol method. You can specify your own protocol designator (such as 'myhttps') if you need to use your custom protocol as well as the default SSL protocol implementation.

```
Protocol.registerProtocol("myhttps",
    new Protocol("https", new MySSLSocketFactory(), 9443));
```

Once registered the protocol be used as a 'virtual' scheme inside target URIs.




```
HttpClient httpClient = new HttpClient();
GetMethod httpget = new GetMethod("myhttps://www.whatever.com/");
try {
    httpClient.executeMethod(httpget);
    System.out.println(httpget.getStatusLine());
} finally {
    httpget.releaseConnection();
}
```

If you want this protocol to represent the default SSL protocol implementation, simply register it under 'https' designator, which will make the protocol object take place of the existing one

```
Protocol.registerProtocol("https",
    new Protocol("https", new MySSLSocketFactory(), 443));
HttpClient httpClient = new HttpClient();
GetMethod httpget = new GetMethod("https://www.whatever.com/");
try {
    httpClient.executeMethod(httpget);
    System.out.println(httpget.getStatusLine());
} finally {
    httpget.releaseConnection();
}
```

Examples of SSL customization in HttpClient

There are several custom socket factories available in our contribution package. They can be a good start for those who seek to tailor the behavior of the HTTPS protocol to the specific needs of their application:

- [EasySSLProtocolSocketFactory](#)  can be used to create SSL connections that allow the target server to authenticate with a self-signed certificate.
- [StrictSSLProtocolSocketFactory](#)  can be used to create SSL connections that can optionally perform host name verification in order to help preventing man-in-the-middle type of attacks.
- [AuthSSLProtocolSocketFactory](#)  can be used to optionally enforce mutual client/server authentication. This is the most flexible implementation of a protocol socket factory. It allows for customization of most, if not all, aspects of the SSL authentication.

Known limitations and problems

1. Persistent SSL connections do not work on Sun's JVMs below 1.4

Due to what appears to be a bug in Sun's older (below 1.4) implementation of Java Virtual Machines or JSSE there's no reliable way of telling if an SSL connection is 'stale' or not. For example, the HTTP 1.1 specification permits HTTP servers in 'keep-alive' mode to drop the connection to the client after a given period inactivity without having to notify the client, effectively rendering such connection unusable or 'stale'. For the HTTP agent written in Java there's no reliable way to test if a connection is 'stale' other than attempting to perform a read on it. However, a read operation on an idle SSL connection on Sun JVM older than 1.4 returns 'end of stream' instead of an expected read timeout. That effectively makes the connection appear 'stale' to HttpClient, which leaves it with no other way but to drop the connection and to open a new one, thus defeating HTTP 1.1 keep-alive mechanism and resulting in significant performance degradation (SSL authentication is a highly time consuming operation). The problem appears to have been fixed in Sun's Java 1.4 SSL implementation. Sockets which are not using HTTPS are unaffected on any JVM.

Workaround: Disable stale connection check if upgrade to Java 1.4 or above is not an option. Please note that HttpClient will no longer be able to detect invalid connections and some requests may fail due to transport errors. For details on how transport errors can be recovered from please refer to the [Exception Handling Guide](#). If persistent SSL connections support and transport reliability is an issue for your application we strongly advise you to upgrade to Java 1.4.

2. Authentication schemes that rely on persistent connection state do not work on Sun's JVMs below 1.4 if SSL is used

This problem is directly related to the problem described above. Certain authentication schemes or certain implementations of standard authentication schemes are connection based, that is, the user authentication is performed once when the connection is being established, rather than every time a request is being processed. Microsoft NTLM scheme and Digest scheme as implemented in Microsoft Proxy and IIS servers are known to fall into this category. If connections cannot be kept alive the user authorization is lost along with the persistent connection state

Workaround: Disable stale connection check or upgrade to Java 1.4 or above.

3. JSSE prior to Java 1.4 incorrectly reports socket timeout.

Prior to Java 1.4, in Sun's JSSE implementation, a read operation that has timed out incorrect reports end of stream condition instead of throwing `java.io.InterruptedIOException` as expected. HttpClient responds to this exception by assuming that the connection was dropped and throws a `NoHttpResponseException`. It should instead report "`java.io.InterruptedIOException: Read timed out`". If you encounter `NoHttpResponseException` when working with an older version of JDK and JSSE, it can be caused by the timeout waiting for data and not by a problem with the connection.

Work-around: One possible solution is to increase the timeout value as the server is taking too long to start sending the response. Alternatively you may choose to upgrade to Java 1.4 or above which does not exhibit this problem.

The problem has been discovered and reported by Daniel C. Amadei.

4. **HttpClient does not work with IBM JSSE shipped with IBM Websphere Application Platform**

Several releases of the IBM JSSE exhibit a bug that cause HttpClient to fail while detecting the size of the socket send buffer (java.net.Socket.getSendBufferSize method throws java.net.SocketException: "Socket closed" exception).

Solution: Make sure that you have all the latest Websphere fix packs applied and IBMJSSE is at least version 1.0.3. HttpClient users have reported that IBM Websphere Application Server versions 4.0.6, 5.0.2.2, 5.1.0 and above do not exhibit this problem.

Troubleshooting

JSSE is prone to configuration problems, especially on older JVMs, which it is not an integral part of. As such, if you do encounter problems with SSL and HttpClient it is important to check that JSSE is correctly installed.

The application below can be used as an ultimate test that can reliably tell if SSL configured properly, as it relies on a plain socket in order to communicate with the target server. If an exception is thrown when executing this code, SSL is not correctly installed and configured. Please refer to Sun's official resources for support or additional details on JSSE configuration.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.net.Socket;

import javax.net.ssl.SSLSocketFactory;

public class Test {

    public static final String TARGET_HTTPS_SERVER = "www.verisign.com";
    public static final int    TARGET_HTTPS_PORT    = 443;

    public static void main(String[] args) throws Exception {

        Socket socket = SSLSocketFactory.getDefault().
            createSocket(TARGET_HTTPS_SERVER, TARGET_HTTPS_PORT);
        try {
            Writer out = new OutputStreamWriter(
                socket.getOutputStream(), "ISO-8859-1");
            out.write("GET / HTTP/1.1\r\n");
            out.write("Host: " + TARGET_HTTPS_SERVER + ":" +
                TARGET_HTTPS_PORT + "\r\n");
            out.write("Agent: SSL-TEST\r\n");
            out.write("\r\n");
            out.flush();
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream(), "ISO-8859-1"));
            String line = null;
            while ((line = in.readLine()) != null) {
                System.out.println(line);
            }
        } finally {
            socket.close();
        }
    }
}
```

