Oracle Blog

Java Platform Group, Product Management blog

Thoughts on Java SE, Java Security and Usability

« Nashorn, the rhino... | Main | Deep monitoring with... »

Diagnosing TLS, SSL, and HTTPS

By Erik Costlow-Oracle on Jul 02, 2014

When building inter-connected applications, developers frequently interact with TLS-enabled protocols like HTTPS. With recent emphasis on encrypted communications, I will cover the way in which the JDK evolves regarding protocols, algorithms, and changes, as well as some advanced diagnostics to better understand TLS connections like HTTPS.

Most developers will not have to do this level of diagnosis in the process of writing or running applications. In the event that you do, the following information should provide enough information to understand what's happening within secure connections.

Stability: The evolution of protocols and algorithms

For the last 15 years (since 1998), the Java platform has evolved through the Java Community Process where companies, organizations, and dedicated individuals develop and vote on specifications to determine what makes up the Java Platform. Much of the efforts are centered on compatibility, like the TCK, ensuring that different implementations are compatible with each-other and that developers can predict how their applications will run. We are not changing critical default options (like TLS protocol) within minor versions.

The following chart depicts the protocols and algorithms supported in each JDK version:

	JDK 8 (March 2014 to present)	JDK 7 (July 2011 to present)	JDK 6 (2006 to end of public updates 2013)

TLS Protocols	TLSv1.2 (default) TLSv1.1 TLSv1 SSLv3	TLSv1.2 TLSv1.2 TLSv1 (default) SSLv3	TLSv1 (default) SSLv3
JSSE Ciphers:	Ciphers in JDK 8	Ciphers in JDK 7	Ciphers in JDK 6
Reference:	JDK 8 JSSE	JDK 7 JSSE	JDK 6 JSSE
Java Cryptography Extension, Unlimited Strength (explained later)	JCE for JDK 8	JCE for JDK 7	JCE for JDK 6

Sample Java code for making an HTTPS connection

Making an HTTPS connection in Java is relatively straight-forward. I will post the code here with the intent focused on tuning and understanding the underlying capabilities.

Sample back-end code for making an SSL connection:

```
final URL url = new URL("https://example.com");
try(final InputStream in = url.openStream()){
  //…
}
```

Or the connection can be tuned through a cast:

```
final HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
//operate on conn
conn.connect();
try(final InputStream in = conn.getInputStream()){
  //…
}
```

Example: Qualys SSL Labs' "View My Client" Page

Qualys SSL Labs maintains a collection of tools that are helpful in understanding SSL/TLS connections. One in particular is a View My Client page, which will display information about the client connection. By integrating with that page, I was able to control the implementation as I used different Java tuning parameters.

To test parameter tuning, I implemented a small JavaFX application in JavaScript. It displays that page in a WebView, showing information about the underlying Java SSL/TLS client connection. You can find the code in the appendix.

JSSE Tuning Parameters

When diagnosing TLS-related issues, there are a number of helpful system properties. They are generally covered in their relevant sections of JSSE but this single collection

may help anyone looking to understand the flexibility of Java's implementation or diagnose connection details.

javax.net.debug	Prints debugging details for connections made. Example: -Djavax.net.debug=all or -Djavax.net.debug=ssl:handshake:verbose
https.protocols	Controls the protocol version used by Java clients. For older versions, this can update the default in case your Java 7 client wants to use TLS 1.2 as its default. Example: -Dhttps.protocols=TLSv1,TLSv1.1,TLSv1.2
http.agent	When initiating connections, Java will apply this as its user-agent string. Modifying this will handle cases where the receiving party responds differently based on the user-agent. Example: -Dhttp.agent="known agent"
java.net.useSystemProxies	java.net.useSystemProxiesUse proxy details from the operating system itself. Example: -Djava.net.useSystemProxies=true
http.proxyHost http.proxyPort	The proxy connection to use for HTTP connections. Example: -Dhttp.proxyHost=proxy.example.com -Dhttp.proxyPort=8080
https.proxyHost https.proxyPort	The same as above, except that configuration is separate between HTTP and HTTPS.
http.proxyUser http.proxyPassword https.proxyUser https.proxyPassword	Password-based credentials for the above proxies.

Many other protocols and properties can be found within the following areas:

- Java Networking and Proxies.
- Java Networking core documentation.
- Java Secure Socket Extension (JSSE) Reference Guide.

Example of diagnosing a problem

When making an HTTPS connection, let's assume that the client threw the following exception due to a failed handshake with the server:

```
javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure
```

SSLHandshakeException is a subclass of the IOException, so you do not need to catch is explicitly. Most developers will not need an explicit catch, but it may help you more easily diagnose the cause of any IOException.

When applying the -Djavax.net.debug=all property from above, the failure associated with this SSLHandshakeException would appear immediately after algorithm negotiation in the logs.

JDK 7 (fails on unsupported algorithm)	JDK 8 (works fine)
Cipher Suites: [...Long list of ciphers...] Compression Methods: { 0 } Extension elliptic_curves, curve names: {...} Extension ec_point_formats, formats: [uncompressed] Extension server_name, server_name: [host_name: HOST] *** main, WRITE: TLSv1 Handshake, length = 168 main, READ: TLSv1 Alert, length = 2 main, RECV TLSv1 ALERT: fatal, handshake_failure main, called closeSocket() main, handling exception: javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure	Cipher Suites: [...Long list of ciphers...] Compression Methods: { 0 } Extension elliptic_curves, curve names: {...} Extension ec_point_formats, formats: [uncompressed] Extension signature_algorithms, signature_algorithms: ... Extension server_name, server_name: [type=host_name (0), value=HOST] *** main, WRITE: TLSv1.2 Handshake, length = 226 main, READ: TLSv1.2 Handshake, length = 89 *** ServerHello, TLSv1.2 RandomCookie: GMT: -1809079139 bytes = { ...} Session ID: {...} Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 Compression Method: 0 Extension renegotiation_info, renegotiated_connection: <empty> Extension ec_point_formats, formats: [uncompressed, ansiX962_compressed_prime, ansiX962_compressed_char2] *** %% Initialized: [Session-1, TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256] ** TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 main, READ: TLSv1.2 Handshake, length = 2308

In the case above, the failure occurred during the handshake. The most likely cause for that is algorithm support. The JDK provides a separate package called JCE Unlimited Strength, designed to add stronger algorithm support than what's available by default. Qualys SSL Labs provides a different server SSL test that will enumerate which algorithms a server supports.

Many TLS error messages are covered in a few pieces of documentation:

- The JSSE Reference Guide's Debugging Utilities contains many commands, sample code, and error message diagnostics.
- Debugging SSL/TLS Connections provides details on how to read the output from using the javax.net.debug options.
- How SSL Works inside the JSSE provides an explanation of the underlying protocols, explaining which messages may occur when something goes wrong.
- A third party book, Bulletproof SSL, contains a chapter on TLS in Java.

Adding stronger algorithms: JCE Unlimited Strength

In a high security environment, one way of strengthening algorithms in the JDK is through the JCE Unlimited Strength policy files. In this particular case, replacing those policy files within JDK 7 allows it to use the stronger variants of existing algorithms and connect successfully.

JCE Unlimited Strength downloads: JDK 8, JDK 7, or JDK 6.

Appendix

The following code will open Qualys SSL Labs' View My Client page within a Java client. To test configurations, run this like:

jjs -fx viewmyclient.js
jjs -fx -Dhttps.protocols=TLSv1 viewmyclient.js

```
var Scene = javafx.scene.Scene;
var WebView = javafx.scene.web.WebView;
var browser = new WebView();
browser.getEngine().load("https://ssllabs.com/ssltest/viewMyClient.html");
$STAGE.scene = new Scene(browser);
$STAGE.show();
```

Category: Oracle

Tags: coding cryptography javascript

Permanent link to this entry

« Nashorn, the rhino... | Main | Deep monitoring with... »
Comments:

> final URL url = new URL("https://example.com");

and

> final HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();

Why You always append final keyword? what does mean?

Posted by **KD** on July 06, 2014 at 02:50 AM PDT #

The final keyword is technically unnecessary here, it's more a habit of my coding style.
Final means that the variable can't be re-assigned later (through an equals).
That way when I come back to read the code a few months later, I don't have to mentally keep track of which variables change. Or if I go to make edits, I don't accidentally change something without understanding the impact.

Posted by **costlow** on July 07, 2014 at 09:34 AM PDT #

I have some questions on DRS from your other blog posts, but they are closed from comments.

Can we email about these? I don't want to clutter this post.

Thanks!

Posted by **guest** on July 14, 2014 at 05:48 AM PDT #

Thank you for asking. On older entries, the comments close after a month or two.
I'd rather keep the comments here directed towards the post topic. If you want to post a question on a site like ServerFault, I can take a look at it.

Posted by **costlow** on July 15, 2014 at 09:58 AM PDT #

Post a Comment:
Comments are closed for this entry.

About

This blog contains topics related to Java SE, Java Security and Usability. The target audience is developers, sysadmins and architects that build, deploy and manage Java applications. Contributions come from the Java SE Product Management team.

Search

Enter search term:

Search only this blog

Recent Posts

- Upcoming Oracle Java SE 7u72 PSU
- Upgrading major Java versions
- Choosing 64 and/or 32 bit Java
- New Management Console in Java SE Advanced 8u20
- Keeping users on Internet Explorer up-to-date and secure
- Java 7 update 67 patch release
- Deep monitoring with JMX
- Diagnosing TLS, SSL, and HTTPS
- Nashorn, the rhino in the room

- Compact Profiles: Space and Security

Top Tags

- adminstrator
- advanced
- annotations
- authentication
- code-sign
- coding
- cryptography
- desktop
- documentation
- drs
- explorer
- internet
- java
- java8
- javaone
- javascript
- jdk6
- jdk7
- jdk8
- liveconnect
- sandbox
- se
- secure
- security
- sysadmin
- upgrade

Categories

- Oracle

Archives

« October 2014

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Today

Bookmarks

- Henrik's Oracle Java SE, ME, Javacard Blog
- The Java Source
- Java on OTN

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /Oracle
- Comments

Atom

- All
- /Oracle
- Comments