JDK9 Jetty ALPN Review

https://pastebin.mozilla.org/8923862

This document presents code notes that are intended to accompany a voice call to review the JDK9 ALPN API as used by a servlet container. Green text indicates what our preferred outcome would be, Red text indicates the current state. Links are to applicable code and/or javadoc.

Negotiating Ciphers

So the first code that I'd like to show you is in way of background to show that negotiating a TLS cipher is a very complex mechanism dependent on private APIs and user extensible behaviour:

1. The SslEngine delegates the choice of cipher to ServerHandshaker#chooseCipherSuite, which iterates over preferred, proposed and legacy ciphers.

2. For each possible cipher the method ServerHandshaker#trySetCipherSuite is called until one returns true. This is a complex method encoded many special cases, using private APIs to extract cipher characteristics and specific algorithm knowledge.

3. Ultimately a specific certificate is chosen by calling either the X509KeyManager#chooseServerAlias or X509ExtendedKeyManager.html#chooseEngineServerAlias methods in a loop, iterating over key type and issuers combinations until an acceptable certificate is returned. This is user-extensible code, where SNI is typically implemented, but users are free to provide their own KeyManager extensions that may apply arbitrary logic to the selection and/or rejection of certificates.

While the behaviour of this negotiation can probably be modelled for the 99% case with much simpler code, it is fundamentally impossible to predict the outcome of cipher negotiation without actually running this cipher negotiation code. The only practicable way to determine which cipher SslEngine will negotiate is by letting an SslEngine process the ClientHello.

Jetty SSLEngine Usage

Jetty's usage of the SSLEngine API is done within the DecryptedEndPoint#fill method. This is within the scope of a SslConnection that is unaware of what protocol will be run over the connection, thus mechanisms like ALPN protocol negotiation are delegated to abstract classes.

We don't need to look at this method in detail other than to give an overview of how we plugin the ALPN behaviour:

- As data is received, up until we set the `_helloPreProcessed` flag, we make a slice of the data buffer and call an abstract ClientHelloProcessor#preProcess so that we can separately parse the ClientHello frame for ALPN extensions. The data is also passed to `SslEngine#unwrap`
- Once `SslEngine#unwrap` has consumed the ClientHello, it goes into the NEED_TASK state; when the task is executed, it performs cipher negotiation (as above).
- Once the cipher is successfully negotiated, the SSLEngine enters the NEED_WRAP state so it can generate the ServerHello frame. Before doing the wrap call, we call an abstract ClientHelloProcessor#postProcess method to act on the negotiated cipher.

In short we break up our ALPN handling into an abstract preProcess(), which looks at the ClientHello before a cipher is negotiated; and an abstract postProcess(), which is called after a cipher is negotiated.

ClientHelloALPNProcessor

The ClientHelloALPNProcessor is written to implement ALPN generically, so it uses a pluggable ALPNServerProvider interface to delegate the ALPN decisions about which application protocols are: available; preferred; suitable for each cipher. Currently we only have a HTTP2 implementation, but it is possible if not expected that other ALPN usages will emerge.

#preProcess()

The preProcess() method is called to process the ClientHello frame extensions externally to the SSLEngine:

- 68: We pass bytes to our own ClientHello parser, which ultimately extracts tlsProtocol, ciphersOffered and protocolsOffered
- 76-78: Once the frame is parsed, we firstly create a list of available ciphers by trimming the enabled cipher with those offered by the client.
- 96-124: For each available cipher, we call an abstract ALPNServerProvider#select method to ask our connection factory which protocol it would select IF this cipher was negotiated. If no protocol would be selected, the cipher is removed from the available ciphers, otherwise we remember the available in a cipher2protocol map.
- 105: Note that the select protocol callback is an extensible. Currently our only implementation is the HTTP2 logic, however in future other ALPN uses may have arbitrary logic here.
- 132: We configure the SslEngine with the trimmed available cipher list, which will be used in the subsequent cipher negotiation.
- 137-139: Ideally here we would have like to configure the SslEngine with the cipher2protocol map, so it could pick which application protocol to use depending on the negotiated cipher. However the API does not support that, so instead we sort the protocols in server priority order and pick the first one and hope it will be usable for the negotiated cipher (but we do check later)!
  It can be debated if picking the application protocol by server priority is best. Perhaps we should sort by server cipher order or even client cipher order? However, whichever way we choose, it is still ultimately a guess that is dependent on the outcome of the cipher negotiation.
  However, even if this API is not changed to allow a map to be set, we could still avoid the guess with the behaviour change described below.

#postProcess()

The postProcess() method is called to check and handle the negotiated cipher:

- 155: This is called after the cipher negotiation, so there is now a SSLSession instance
- 160: We check the same TLS protocol was negotiated
- 164: We discover which cipher was negotiated
- 165: We lookup what application protocol should be used with that cipher.
- 170: If the application protocol configured in preProcess() is not the same as the protocol for the negotiated cipher then:
    - 175-176: We set the SSLEngine application protocol to the protocol. This is before the wrap that generates the ServerHello frame, so it should be able to be used. However the current implementation of SSLEngine ignores this call after the ClientHello has been received. See SSLEngineImpl#setSSLParameters. The ServerHandshaker.started() method returns the value of the Handshaker.clientHelloDelivered field, which is set at Handshaker.java#processLoop, *before* the processing of the ClientHello frame actually happens. Note that if the call to SSLEngine.setSSLParameters() was not ignored, then we would change the Jetty code to not guess in preProcess() and always just set the application protocol here in postProcess(), after cipher negotiation.
    - 178-183: Instead we will need to: create a new SSLEngine instance; configure it with the cipher and protocol; replay the ClientHello message (which will have to be saved in preProcess()); and then replace the SSLEngine in our calling scope with the new one!. We have not yet implemented this code as it is moderately invasive and best to be avoided if we can.
- 187: Signal the choice of application protocol. In the case of HTTP, this allows our ConnectionFactory to provide a connection implementing the negotiated version of the protocol. The protocol connection so constructed is then wrapped by the existing SslConnection.

Summary

We do understand that the use-case we are concerned about is small, and that for most modern browsers talking to a server running on a recent JVM, that a HTTP2 cipher will be successfully negotiated and thus the server guess will be right.

However, any server that has h2 acceptable certificates for only some of SNI hosts will be vulnerable to making a wrong guess about the application protocol. Moreover, future usages of ALPN may follow h2's example and introduce arbitrary logic that makes the negotiated application protocol a function of the negotiated cipher.

Thus the use-case is non-zero now and possible may be even more used in future.

Currently when the application protocol guess is wrong, significant extra effort/resources are required to correct the guess. Many implementation will probably not handle this case and thus connection failure will result.

The solution is to avoid guessing an application protocol, which can be done by either:

1. Changing the API so a cipher to protocol map can be set before processing the ClientHello. OR
2. Changing the SSLEngine implementation so that the application protocol can be changed after the cipher is negotiated, but before the ServerHello frame is generated.