State of security in JDK 11

The Java security sandbox model has been around since more than a decade and with some enhancements over years remains pretty much the same as of JDK 11. On the other hand the JDK is bringing a number of security utilities for use by developers such as:

- JCA (Java Cryptography Architecture)
- PKI (Public Key Infrastructure) utilities
- JSSE (Java Secure Socket Extension)
- Java GSS API (Java Generic Security Services)
- Java SASL API (Java Simple Authentication and Security Layer)

All of the above have also been around for some time and enhanced throughout the JDK versions. Although these utilities provide an abundance of options for implementing security in a Java application developers still tend to choose in addition a number of third-party security libraries and frameworks that provide alternative or missing capabilities compared to those provided by the JDK (such as BouncyCastle for enhanced cryptography algorithms and utilities or JSch for SSH to name a few). In this article we will look into the state of JDK security as of JDK 11.

JDK 11 from a security perspective

Major enhancements have been introduced in the JSSE API in regard to the TLS support in the JDK. These include:

- DTLS – the introduction of a **datagram transport layer security protocol** in the JDK enables applications to establish secure communication over an unreliable protocol such as UDP;

- ALPN – the **application layer protocol negotiation** extension of TLS enable applications to negotiate the application protocol for use by communicating parties during the TLS handshake that establishes the secure channel. Consider for example HTTP 1.1 and HTTP 2.0: a TLS client and server may negotiate which version of the HTTP protocol to use during the TLS handshake process;

- OCSP stapling – allows for certificate revocation checking requests from the TLS client to the certificate authority that use the OCSP protocol to be performed by the TLS server thus minimizing the number of requests to the certificate authority.

All of the above are introduced in JDK 9 while in JDK 10 a default set of root certificate authorities have been provided to the trust store of the JDK for use by Java applications. Moving further indisputably the security highlight of JDK 11 is the implementation of major parts of the TLS 1.3 protocol in JSSE. Major benefits of TLS 1.3 are improved security and performance including the following enhancements introduced by the protocol (as highlighted by JEP 332 upon which the implementation is based):

- Enhanced protocol version negotiation using an extension field indicating the list of supported versions;
- Improved full TLS handshake for both client and server sides (more compact than in TLS 1.3);
- Improved session resumption using a PSK (Pre-Shared Key) extension;
- The possibility to update cryptographic keys and corresponding initialization vectors (IV) after a Finish request is send in the TLS handshake;
- Additional extensions and algorithms;
- Two new cipher suites: TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384
- RSASSA-PSS signature algorithms;

It is good to note that TLS 1.3 is not backward compatible with previous versions of the protocol but the JSSE implementation provides backward compatibility mode. In addition both the synchronous (via the SSLSocket API) and asynchronous mode of operation (via the SSLEngine API) are updated to support TLS 1.3. The following example shows a JSSE synchronous SSL server and client using TLS 1.3 (note that the examples need to be run using JDK 11):

TLSv1.3 JSSE server

```
System.setProperty("javax.net.ssl.keyStore", "C: /sample.pfx");
System.setProperty("javax.net.ssl.keyStorePassword", "sample");

SSLServerSocketFactory ssf = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
  SSLServerSocket ss = (SSLServerSocket) ssf.createServerSocket(4444);
  ss.setEnabledProtocols(new String[] {"TLSv1.3"});
  ss.setEnabledCipherSuites(new String[] {"TLS_AES_128_GCM_SHA256"});

  while (true) {
   SSLSocket s = (SSLSocket) ss.accept();
   SSLParameters params = s.getSSLParameters();
   s.setSSLParameters(params);

   BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
   String line = null;
   PrintStream out = new PrintStream(s.getOutputStream());
   while (((line = in.readLine()) != null)) {
    System.out.println(line);
        out.println("Hi, client");
   }
   in.close();
   out.close();
```

```
            s.close();
        }
```

TLSv1.3 JSSE client

```
        LOGGER.info("Sending message via TLSv1.3 ... ");

        try {

                System.setProperty("javax.net.ssl.trustStore", "C:/sample.pfx");
                System.setProperty("javax.net.ssl.trustStorePassword", "sample");

                SSLSocketFactory ssf = (SSLSocketFactory) SSLSocketFactory.getDefault();
                SSLSocket s = (SSLSocket) ssf.createSocket("127.0.0.1", 4444);
                s.setEnabledProtocols(new String[] {"TLSv1.3"});
                s.setEnabledCipherSuites(new String[] {"TLS_AES_128_GCM_SHA256"});

                SSLParameters params = s.getSSLParameters();
                s.setSSLParameters(params);

                PrintWriter out = new PrintWriter(s.getOutputStream(), true);
                out.println("Hi, server.");
                BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
                String x = in.readLine();
                System.out.println(x);

                out.close();
                in.close();
                s.close();
        } catch (Exception ex) {
                ex.printStackTrace();
        }
```

As you can see from the highlighted lines above in order to enable TLS 1.3 in JDK 11 we need to specify the appropriate constant representing the protocol version over the server and client SSL sockets created and also set the appropriate cypher suites that TLS 1.3 expects again on both the client and server SSL sockets.

Security Sandbox Model: A brief recap

The security sandbox remains the same as of JDK 9 where Jigsaw modules bring subtle changes to the JDK so that the permission model can be applied to modules. This is achieved by simply introducing a new scheme (**jrt**) for referring to Jigsaw modules when specifying permissions in the security.policy file.

For example if we install a security manager in the JSSE server we created by adding the following at the beginning of the source code:

```
System.setSecurityManager(new SecurityManager());
```

When you rerun the JSSE server you will get:

```
java.security.AccessControlException: access denied ("java.util.PropertyPermission"
"javax.net.ssl.trustStore" "write")
```

This is simply because we also need to put the proper permissions in the security.policy file residing in the JDK installation directory (by default that is under conf/security) for the JDK we use to run the JSSE server. If the codebase (location from where we start the JSSE server) is the compilation directory (i.e. we run the JSSE server from the compiled Java class containing the snippet) in the security.policy file we would end up with (adding a few more permissions required):

```
grant codeBase "file:/C:/project/target/" {
                permission java.util.PropertyPermission "javax.net.ssl.keyStore", "write";
                permission java.util.PropertyPermission "javax.net.ssl.keyStorePassword", "write";
                permission java.net.SocketPermission "localhost:4444", "listen,resolve";
};
```

If our JSSE server was packaged as a JDK module named “com.exoscale.jsse.server” we could have specified the above entry in the following format:

```
grant codeBase "jrt:/com.exoscale.jsse.server " {
                …
};
```

Deploying the sample application

We are going to demonstrate is manual deploy which of course can be automated with proper tools. Assuming you have provisioned an Linux Ubuntu 18.04 LTS 64-bit (i.e. from the Exoscale Web UI) you can ssh to the machine and do the following (assuming we have bundled our application in a runnable JAR and having it uploaded it somewhere accessible from your VM):

```
sudo apt-get update
sudo add-apt-repository ppa:linuxuprising/java
sudo apt-get install oracle-java11-installer
wget https://filebin.net/vxsv072m3jebv90m/jsseserver.jar?t=ba3n293q –O jsseserver.jar
java –jar jsseserver.jar
```

Of course you need to also specify the proper permissions in the security.policy file of the installed JDK in case you run your JSSE server with a security manager installed.

Conclusion

We saw briefly how to get started with TLS 1.3 in JDK 11, how to further apply the JDK security sandbox model over your application and how to provision your JDK application on the Exoscale cloud. Further JDK versions will continue providing security improvements.