TLS

Table of Contents

Quick Start 1
Production Setup 6
Production Setup 7

In order to configure TLS you will have to obtain a public/private key, a X.509 certificate, add those to the Java keystore and optionally add certificates from a known CA (certicate authority). The entire process can be confusing but in order to get a basic setup for testing purposes up and running with minimal effort, this section starts off with a simple quick start. However, for production environment you need to obtain an officially signed certificate from a known CA and that process is outlined in section Production Setup.

Quick Start

This section shows how to create a self signed certificate, how to add that to the Java keystore and how to configure the SIP Servlet Container to make use of this configuration. Note, this section should only be used in a development environment and the main reason for this quickstart section is to get you going right away as well as get you comfortable with generating keys and certificates and adding them to the Java keystore.

Procedure: Server Side Authentication

At a high-level, we will execute the following three steps:

1. Generate a public/private key pair and a self signed certificate and add those to the Java keystore.
2. Configure the SIP Servlet Container to load our certificate from the keystore.
3. Test!

Let's follow each step in order:

1. Generate certificate

   Generating a new key-pair and a certificate can be done in a few different ways with a few different tools but here we will just use the java keytool that comes with the JDK. Simple issue the following command, which will generate a new public and private key, generate a self-signed certificate and add it all to the Java keystore:

   ```
   keytool -genkeypair -alias myserver -keyalg RSA -keysize 1024 -keypass secret
   -validity 365 -storetype jks -keystore myserver.jks -storepass secret -v -dname
   "CN=James Smith, OU=Engineering, O=My Company, L=My City, S=My State, C=US"
   ```

   -keystore specifies which keystore we should use/update. If the keystore doesn't exist, a new one will be created for one. In the above example, we named the keystore `myserver.jks` and it will be saved in the current directory

   -keypass and -storepass should be chosen wisely since with bad passwords you won't have much protection anyway. Also, normally you should never passwords on the command prompt, it is too easy for other people to steal. If you leave these two options out, the keytool command will ask you for it.

-keyalg specifies which algorithm to use when generating the keys and the keysize how long those keys should be.

Note: the command -genkeypair is new in JDK 6 and was previously named -genkey. The keytool in JDK 6 has some improvements over the previous versions so it is recommended to use it instead.

See more about the Java keytool here: http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html

2. Configure the SIP Servlet Container

The SIP Servlet Container relies on the JAIN SIP stack to support it with TLS capabilities. As such, it is the JAIN SIP stack that we need to configure to have it read our certificate we added to the key store. The various configuration options are described in the javadoc of the SipStackImpl class but for this quickstart, we will be using the following ones:

- `javax.net.ssl.keyStore` – the filename and location of the keystore to use.
- `javax.net.ssl.keyStorePassword` – the password to the keystore.
- `javax.net.ssl.trustStore` – the filename and location of the truststore to use.
- `javax.net.ssl.trustStorePassword` – the password to the truststore.
- `gov.nist.javax.sip.TLS_CLIENT_AUTH_TYPE` – which type of authentication we will require of the client (for now, the client authentication type will be set to Disabled).
- `gov.nist.javax.sip.gov.nist.javax.sip.ENABLED_CIPHER_SUITES` – Comma-separated list of suites to use when creating outgoing TLS connections. This parameter is optional with default value "TLS_RSA_WITH_AES_128_CBC_SHA,SSL_RSA_WITH_3DES_EDE_CBC_SHA,TLS_DH_anon_WITH_AES_128_CBC_SHA,SSL_DH_anon_WITH_3DES_EDE_CBC_SHA"

The configuration options are JVM parameters and you will have to add these to the command line when you start the server:

+

```
./bin/run.sh -Djavax.net.ssl.keyStorePassword=mysecret
-Dgov.nist.javax.sip.TLS_CLIENT_AUTH_TYPE=Disabled
-Djavax.net.ssl.keyStore=/path/to/your/keystore/myserver.jks
-Djavax.net.ssl.trustStorePassword=mysecret
-Djavax.net.ssl.trustStore=/path/to/your/keystore/myserver.jks
```

Once the server is up, we are ready to verify that we can get a TLS connection using the certificate we previously added in the first step.

+

for this first part of the quickstart we will not require a certificate from the client since this involves more configuration. This is controlled by the `gov.nist.javax.sip.TLS_CLIENT_AUTH_TYPE parameter`.

3. Test!

   To verify your setup there are a few different tools that you can use.

   - openssl is an open source SSL toolkit and contains a generic SSL/TLS test client
   - SIPp – an open source SIP load testing tool that is capable of using TLS. However, it requires some additional steps that we have not addressed in the first parf of this quickstart so therefore we willl not be using SIPp.
   - Using your favorite SIP client. Most SIP clients out there are capable of establishing a TLS connection but you will have to consult its documentation of how to configure TLS.

     Using openssl:

     Assuming that your server is running on localhost and is listening for TLS on port 5081 the command would be:

     ```
     openssl s_client -host 127.0.0.1 -port 5081
     ```

     If you are successful you should see an output from openssl displaying information about the server certificate (which should be the one we generated in Step 1). If there are any issues with the setup, openssl is pretty good about giving out information about what it thinks is wrong.

     Tip: if you add the following JVM parameter as well you will get a lot of useful debug information: `-Djavax.net.debug=ssl`

Procedure: Server Side Authentication

In the first part of this quickstart we generated a public and private key along with a self-signed certificate and added them all into the Java keystore. The server was then configured to use this information and when a client connected, our certificate was served up to the client. However, normally, the client and the server would like to verify each others certificate to make sure they both trust each other and if not, either of them will terminate the connection. In the first part of the quickstart, the server did not require the client to present a certificate when connecting (remember that we set the `gov.nist.javax.sip.TLS_CLIENT_AUTH_TYPE` to disabled) so let's do that now.

At a high-level, these are the tasks we need to execute:

1. Generate a public/private key pair for the client along with a certificate.
2. The server need to add the client certificate to its keystore as a trusted certificate.
3. Start the server with client authenticating enabled.

Let's follow each step in order:

1. Generate Client Certificate

   We will use the Java keytool for this step in the same we did for for the server side in the previous quikstart. The command is exactly the same and the only difference is that we store the information in a new keystore called `myclient.jks`.

   ```
   keytool -genkeypair -alias myclient -keyalg RSA -keysize 1024 -keypass secret
   -validity 365 -storetype jks -keystore myclient.jks -storepass secret -v -dname
   "CN=John Doe, OU=Engineering, O=Some Work, L=Some City, S=Some State, C=US"
   ```

   We have now generated a new keystore containing the clients authentication information. However, the server needs to import the client certificate into its trusted keystore so we need to extract the certificate out of the client key store. This can also be done using the Java keytool.

   ```
   keytool -exportcert -alias myclient -file client.cert -keystore myclient.jks
   -storepass secret -rfc
   ```

   The certificate is saved in file 'client.cert' and we will use this file in the next step.

2. Re-configure the server

   Simply change the `gov.nist.javax.sip.TLS_CLIENT_AUTH_TYPE` from 'Disabled' to 'Enabled' and start the server again.

3. Test

   We will once again use openssl to verify our setup but now that the client will be forced to present a certificate as well, we do need the certificate's private key as well. The private key is embedded into the keystore and was generated when we issued the 'kenkeypair' keytool-command. Unfortunately, the keytool does not have an option for exporting the private key so we will have to write a small java program to extract it for us. Luckily, it is not a lot of code:

```
import java.io.FileInputStream;
    import java.security.Key;
    import java.security.KeyStore;
    import sun.misc.BASE64Encoder;

    /**
     * Code originally posted on Sun's developer forums but
     * can now only be found at stackoverflow:
     * http://stackoverflow.com/questions/150167/how-do-i-list-export-private-keys-
from-a-keystore
     */
    public class DumpPrivateKey {

        static public void main(String[] args)
        throws Exception {
            if(args.length < 3) {
            throw new IllegalArgumentException("expected args: Keystore filename,
Keystore password, alias, <key password: default same than keystore");
            }
            final String keystoreName = args[0];
            final String keystorePassword = args[1];
            final String alias = args[2];
            final String keyPassword = getKeyPassword(args,keystorePassword);
            KeyStore ks = KeyStore.getInstance("jks");
            ks.load(new FileInputStream(keystoreName),
keystorePassword.toCharArray());
            Key key = ks.getKey(alias, keyPassword.toCharArray());
            String b64 = new BASE64Encoder().encode(key.getEncoded());
            System.out.println("-----BEGIN PRIVATE KEY-----");
            System.out.println(b64);
            System.out.println("-----END PRIVATE KEY-----");
        }

        private static String getKeyPassword(final String[] args, final String
keystorePassword)
        {
        String keyPassword = keystorePassword; // default case
        if(args.length == 4) {
            keyPassword = args[3];
        }
        return keyPassword;
        }
        }
```

Copy and paste the above code into a file call DumpPrivateKey.java and then compile it:

```
javac DumpPrivateKey.java
```

and then use it to extract the private key:

```
java DumpPrivateKey myclient.jks secret myclient > clientprivate.key
```

Now that we have the private key of the client we can use openssl to verify the setup again:

```
openssl s_client -host 127.0.0.1 -port 5081 -cert client.cert -certform PEM -key
clientprivate.key
```

If all goes well you should successfully establish a connection and openssl will dump information about the certificate exchange.

Production Setup

In a production environment it is important that you run with an officially signed certificate from a known CA. It is this certificate that you will load into your keystore and the process is very similar to the one outlined in the quick start.

1. Generate a PKCS#12 Storage

   Assuming that you already have a private key and a signed certificate from a known CA you first have to wrap these two into a pkcs#12 storage (pkcs#12 is a file format for storing X.509 public certificates along with the private key), and then load that into the Java keystore. To create a pkcs#12 storage you can use the openssl pkcs12 command:

   ```
   openssl pkcs12 -inkey myprivate.key -in mycertificate.pem -export -out
   mystorage.pkcs12 -passout mysecret
   ```

   where myprivate.key is the private key, `mycertificate.pem` is the X.509 certificate. The password for the storage is 'mysecret' and the name of the storage file is `mystorage.pkcs12`.

2. Generate the Java Keystore

   Once the pkcs#12 has been created, use the Java keytool to load the pkcs12 storage and convert it into a java keystore.

   ```
   keytool -importkeystore -srckeystore mystorage.pkcs12 -srcstoretype PKCS12
   -destkeystore myserver.jks -deststorepass mysecret -srcstorepass mysecret
   ```

   A few things to point out:

   -srcstoretype is important and tells the Java keytool which format the key store that we are importing is in. In the previous step, we generated a pkcs#12 store so in this example, the store type must be PKCS12.

-srcstorepass is the password for the pkcs#12 storage and in the above example it is the same as the destination key store (-deststorepass) but most likely they will be different.

3. Re-configure and Test

   Now that we have a java keystore the server configuration is exactly the same as described in the quick start, i.e., simply set the java properties `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` to point to this key keystore file and then set the password through the property `javax.net.ssl.keyStorePassword` and `javax.net.ssl.trustStorePassword`. Once the server has been re-started you can use openssl to verify the setup.

Production Setup

In addition to securing your SIP TLS, you may want to secure your HTTPS and SIP Over WebSockets Connectors too.

1. Secure HTTPS on JBoss 7/EAP 6

   Assuming that you already followed the previous steps, you now have a private key and a self signed certificate. You will need to configure your `$JBOSS_HOME/standalone/configuration/standalone-sip.xml` to enable HTTPS connector:

```
            <subsystem xmlns="urn:jboss:domain:web:1.4" default-virtual-
server="default-host" native="false">
            <connector name="http" protocol="HTTP/1.1" scheme="http" socket-
binding="http"/>
        <connector name="https" protocol="HTTP/1.1" scheme="https" socket-
binding="https" secure="true">
        <ssl protocol="TLSv1,TLSv1.1,TLSv1.2" certificate-key-
file="/path/to/myserver.jks" certificate-file="/path/to/myserver.jks"
password="secret"/>
        </connector>
```

2. Add SIP Over WebSockets Secure Connector

   Make sure the following connector is present in `$JBOSS_HOME/standalone/configuration/standalone-sip.xml`

```
<connector name="sip-wss" protocol="SIP/2.0" scheme="sip" socket-binding="sip-
wss"/>
```

   Make sure the following socket-binding is present in `$JBOSS_HOME/standalone/configuration/standalone-sip.xml`

```
<socket-binding name="sip-wss" port="5083"/>.
```

3. For self-signed certificates, import the pkcs file to your Browser

   To make that the WebSockets connection is not refused with a self-signed certificate, you need to import the pkcs file generated in 7.2.2 to Google Chrome (Settings ⇒ Show Advanced Settings ⇒ Manage Certificates Button, then import your mystorage.pkcs12 file) or Firefox.

4. Test!

   Go to your WebRTC favorite example through https://localhost:8443/webrtc/, and use wss://localhost:5083 to connect over Secure SIP Over WebSockets.