TLS

Table of Contents

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

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.
4. 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.

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

. 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:
+
[source,xml]
```

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:
+
[source]
```

javac DumpPrivateKey.java

```
+
and then use it to extract the private key:
+
[source]
```

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:
+
[source]
```

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.


[[_sss_tls_production_setup]]
== 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.


. 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
http://www.openssl.org/docs/apps/pkcs12.html[openssl pkcs12] command:
+

[source]
```

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

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

. 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.
+

[source]
```

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.


. 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.
```