BlogtimeException

Implementing mutual authentication over SSL in Java

Behrang Saeedzadeh Jan 30th, 2019

A common practice in relatively large organizations is to secure their internal APIs using SSL key pairs issued by their own private CAs. In a previous article we created a private CA and used it to issue two sets of key pairs. In this article we put those key pairs into use:

- First, we use the key pair we had created for `*.tmnt.local` to secure a Tomcat server.
- Then, we write a Java program to connect to our Tomcat server over HTTPS.
- After that we modify our Tomcat server to require client authentication.
- And finally we modify our Java program to present the public certificate we had created for Donatello during handshake and authenticate itself to the server.

This setup in which both the client and the server require each other to identify themselves by sharing their public certificates during SSL handshake is sometimes referred to as *"mutual authentication over SSL"*.

Download Tomcat

First let's download a Tomcat distribution from tomcat.apache.org and extract the archive into a directory on our system. For this example we will go ahead with Tomcat 7.0.92 but the instructions here should work with the 8.x and 9.x versions too.

Bundling tmnt.local's public certificate, private key, and intermediate certificate into a PKCS12 file

To make Tomcat serve content over HTTPS we should bundle the private key and public key for `*.tmnt.local` plus the intermediate certificate of TMNT into a PKCS12 file:

```
$ openssl pkcs12 -export \
  -in tmnt.local.cert.pem \
  -inkey tmnt.local.key.pem \
  -out tmnt.local.p12 \
  -name "tomcat" \
  -certfile intermediate.cert.pem
```

Update Tomcat configuration

Then we should tweak Tomcat's `server.xml` and enable HTTPS. The snippet below configures an HTTPS connector on port 8443 that is secured using the `*.tmnt.local` key pair:

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="8443"
  maxThreads="200"
  scheme="https"
  secure="true"
  SSLEnabled="true"
  keystoreFile="${user.home}/tmnt.local.p12"
  keystorePass="changeit"
  sslProtocol="TLS"
  clientAuth="false" />
```

Add it to `server.xml` inside the `<Service name="Catalina">` element and restart your Tomcat server for it to take effect.

Edit your computer's /etc/hosts file

To test our setup locally, we can edit our system's `/etc/hosts` file and configure it to route `test1.tmnt.local` to `127.0.0.1` – our computer's private IP address:

```
$ cat /etc/hosts
127.0.0.1  localhost
127.0.0.1  test1.tmnt.local
```

Write a Java program to read from https://test1.tmnt.local:8443

Let's write a simple Java program that connects to `https://test1.tmnt.local:8443` and dumps it content on the screen:

```
package org.behrang.examples.massl;

import java.net.URL;

public class Main {
  public static void main(String[] args) throws Exception {
    var url = new URL("https://test1.tmnt.local:8443/");
    try (var urlInput = url.openStream()) {
      System.out.println(new String(urlInput.readAllBytes()));
    }
  }
}
```

If we run it, it fails with this error message:

If we run it, it fails with this error message:

```
Exception in thread "main" javax.net.ssl.SSLHandshakeException:
  sun.security.validator.ValidatorException:
    PKIX path building failed:
      sun.security.provider.certpath.SunCertPathBuilderException:
        unable to find valid certification path to requested target
```

This happens because Java's default trust store file does not trust our custom CA. To fix this error we should create a custom trust store that contains TMNT's root CA certificate and configure our program to use that instead of Java's default trust store.

Create a trust store containing TMNT's root CA certificate

We can use keytool to create our custom trust store:

```
$ keytool -import \
  -file root.cert.pem \
  -alias tmnt-root \
  -keystore tmnt-truststore.jks
```

Let's rerun our program and configure it to use `tmnt-truststore` as its trust store:

```
package org.behrang.examples.massl;

import java.net.URL;

public class Main {
  public static void main(String[] args) throws Exception {
    System.setProperty("javax.net.ssl.trustStore", "/path/to/tmnt-truststo

    var url = new URL("https://test1.tmnt.local:8443/");
    try (var urlInput = url.openStream()) {
      System.out.println(new String(urlInput.readAllBytes()));
    }
  }
}
```

This time, the connection is established successfully and the content is fetched and printed on the screen without any errors.

Configure Tomcat to require client authentication

Modify the `Connector` element that we added earlier to `server.xml`, set `clientAuth` to `true` and also add these two extra attributes to it:

```
<Connector
   ...
  truststoreFile="/path/to/tmnt-truststore"
  truststorePass="changeit"
   ...
>
```

It should now look like this:

```
<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"
  port="8443"
  maxThreads="200"
  scheme="https"
  secure="true"
  SSLEnabled="true"
  keystoreFile="${user.home}/tmnt.local.p12"
  keystorePass="changeit"
  sslProtocol="TLS"
  truststoreFile="${user.home}/tmnt-truststore"
  truststorePass="changeit"
  clientAuth="true" />
```

Now Tomcat will trust any client certificates that are signed by TMNT's CA. Restart Tomcat and rerun the app. The app will now produce another exception:

```
Exception in thread "main" javax.net.ssl.SSLHandshakeException:
  Received fatal alert: bad_certificate
```

This error happens because our server is expecting the client to present its public certificate during handshake but our Java program is not configured to do that. To fix this, first let's bundle Donatello's key pair into another PKCS12 file:

```
$ openssl pkcs12 -export \
  -in donatello.cert.pem \
  -inkey donatello.key.pem \
```

```
  -out donatello.p12 \


  -name "donatello" \
  -certfile intermediate.cert.pem
```

Now we configure our Java program to use `donatello.p12` as its key store:

```
package org.behrang.examples.massl;

import java.net.URL;

public class Main {
  public static void main(String[] args) throws Exception {
    System.setProperty("javax.net.ssl.trustStore", "/path/to/tmnt-truststo
    System.setProperty("javax.net.ssl.keyStore", "/path/to/donatello.p12")
    System.setProperty("javax.net.ssl.keyStoreType", "pkcs12");
    System.setProperty("javax.net.ssl.keyStorePassword", "donatellopass");

    var url = new URL("https://test1.tmnt.local:8443/");
    try (var urlInput = url.openStream()) {
      System.out.println(new String(urlInput.readAllBytes()));
    }
  }
}
```

If we rerun the app, it will successfully connect to `test1.tmnt.local` over HTTPS and fetch and print the content once again.

Passing system properties via the command line

Of course, we can pass the system properties as command line arguments. First, let's remove the calls to `System.setProperty` from our program:

```
package org.behrang.examples.massl;

import java.net.URL;

public class Main {
  public static void main(String[] args) throws Exception {
    var url = new URL("https://test1.tmnt.local:8443/");
    try (var urlInput = url.openStream()) {
```

```
      System.out.println(new String(urlInput.readAllBytes()));
    }

  }
}
```

Now after recompiling our program we can pass these custom properties via the CLI:

```
$ java -Djavax.net.ssl.trustStore=/path/to/tmnt-truststore \
  -Djavax.net.ssl.keyStore=/path/to/donatello.p12 \
  -Djavax.net.ssl.keyStoreType=pkcs12 \
  -Djavax.net.ssl.keyStorePassword=donatellopass \
  -cp /path/to/examples-massl-1.0-SNAPSHOT.jar \
  org.behrang.examples.massl.Main
```

This will, again, successfully connect to `test1.tmnt.local` and print its content on the console.

Creating our program's trust store and key store programmatically

Another option is to create our trust store and key store programmatically, instead of using custom `javax.net.ssl.*` system properties. In some scenarios, for example when your program should connect to multiple HTTPS endpoints each of which expecting certificates signed with a different CA, this is the option to choose:

```
package org.behrang.examples.massl;

import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManagerFactory;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.URL;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;

public class Main {
  static KeyManagerFactory buildKeyManagerFactory() throws UnrecoverableKe
```

```
                                                                NoSuchAlgorithm
                                                                KeyStoreExcepti

                                                                IOException,
                                                                CertificateExce
  var storeType = "pkcs12";
  var keyStore = KeyStore.getInstance(storeType);

  var storePass = "donatellopass".toCharArray();
  var storePath = "/path/to/donatello.p12";
  try (var fis = new FileInputStream(storePath)) {
    keyStore.load(fis, storePass);
  }

  var keyManagerFactory = KeyManagerFactory.getInstance(
      KeyManagerFactory.getDefaultAlgorithm()
  );

  var keyPass = "donatellopass".toCharArray();
  keyManagerFactory.init(keyStore, keyPass);

  return keyManagerFactory;
}


static TrustManagerFactory buildTrustManagerFactory() throws KeyStoreExc
                                                             IOException
                                                             NoSuchAlgor
                                                             Certificate
  var storeType = "jks";
  var trustStore = KeyStore.getInstance(storeType);

  var storePath = "/path/to/tmnt-truststore";
  try (var fis = new FileInputStream(storePath)) {
    trustStore.load(fis, null);
  }

  var trustManagerFactory = TrustManagerFactory.getInstance(
      TrustManagerFactory.getDefaultAlgorithm()
  );
```

```
    trustManagerFactory.init(trustStore);

    return trustManagerFactory;
  }

  static SSLContext buildSslContext(
      KeyManagerFactory keyManagerFactory,
      TrustManagerFactory trustManagerFactory) throws KeyManagementExcepti
                                                      NoSuchAlgorithmExcep
    var sslContext = SSLContext.getInstance("TLSv1.3");

    sslContext.init(
        keyManagerFactory.getKeyManagers(),
        trustManagerFactory.getTrustManagers(),
        null
    );

    return sslContext;
  }

  public static void main(String[] args) throws Exception {
    var keyManagerFactory = buildKeyManagerFactory();

    var trustManagerFactory = buildTrustManagerFactory();

    var sslContext = buildSslContext(keyManagerFactory, trustManagerFactor

    var url = new URL("https://test1.tmnt.local:8443/");
    var urlConnection = (HttpsURLConnection) url.openConnection();
    urlConnection.setSSLSocketFactory(sslContext.getSocketFactory());

    try (var urlInput = urlConnection.getInputStream()) {
      System.out.println(new String(urlInput.readAllBytes()));
    }
  }
}
```

Using Apache HttpComponents

Connecting to and getting data from HTTPS URLs by using `URLConnection` and creating trust stores

and key stores manually will teach us how Java handles communication over HTTPS under the hood, but for connecting to REST endpoints it is better to use libraries with cleaner APIs and higher levels of abstraction.

Apache's HttpComponents Client is one of the many neat HTTP client libraries that are available in the Javaverse.

The next example shows how we can enable mutual authentication over SSL and send an HTTPS request to our Tomcat server using Apache HttpComponents HttpClient:

```
package org.behrang.examples.massl;

import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.ssl.SSLContexts;
import org.apache.http.util.EntityUtils;

import java.io.File;

public class Main {
  public static void main(String[] args) throws Exception {
    var sslContext = SSLContexts.custom()
        .loadKeyMaterial(
            new File("/path/to/donatello.p12"),
            "donatellopass".toCharArray(),
            "donatellopass".toCharArray()
        )
        .loadTrustMaterial(
            new File("/path/to/tmnt-truststore")
        )
        .build();

    var httpClient = HttpClients.custom()
        .setSSLContext(sslContext)
        .build();

    var getHomepage = new HttpGet("https://test1.tmnt.local:8443/");
    var response = httpClient.execute(getHomepage);

    System.out.println(EntityUtils.toString(response.getEntity()));
  }
}
```

Wrap up

This article is probably going to be the latest in these series for the near future. We have covered a lot – creating CAs, securing Tomcat, and establishing mutual authenticaion over SSL – but we have barely scratched the surface. Later I will come back and edit these articles and cover more fundamentals.

BlogtimeException
Personal weblog and home page of Behrang Saeedzadeh.

LinkedIn
Twitter
GitHub
StackOverflow

“Begin at once to live, and count each separate day as a separate life.” — Seneca