CS 594—Spring 2019—Enterprise and Cloud Security
Assignment Four—Secure REST Web Service

In this assignment, you will build on some of the infrastructure that you developed for the previous assignment, to implement a secure REST Web Service that uses client certificates for authentication and authorization of Web service clients. **Your client only needs to support the operation of posting a message.**

You are provided with three additional Maven projects (children of the root Maven project):

1. `ChatRepresentations:` Provides Java classes for representations that are exchanged by the Web service.
2. `ChatRestWebService`: A JAX/RS-based Web service that provides access to the functionality of the chat application that you completed and secured for Assignment 2, including resources for getting a list of the messages that have been posted, posting a new message, and deleting an existing message.
3. `ChatRestWebClient`: A JAX/RS-based (actually Jersey-based) Web service client that provides access to this Web service via a desktop application.

You should make this application secure in the following ways.

First, the service needs to authenticate itself to the clients. It will do this by providing an SSL certificate chain with a root signed by a trusted CA. In this case, the intention is that this will be an enterprise application with its own internal CA, where the clients include in their trust stores the self-signed root certificate for this trusted CA. The service must be able to authenticate itself to using a signing key that is certificated by your enterprise root CA. This signing key certificate, and any other certificates to be provided to clients, must be in the key store for the **application server** (which should be of type JKS). The clients only need to have the root certificate (for the root CA) in their trust stores. All other certification that are required to authenticate the service should be provided by the service (application server).

Second, the client needs to authenticate itself to the service. There are two options for doing this:

1. The easiest solution is to use basic Web authentication: The client provides its credentials, in terms of a user name and password, in every Web request. These are passed in the Authorization HTTP request header of every request. The server uses a security realm, such as a JDBC realm, to authenticate a client based on the credentials that they provide.
2. A more satisfactory is to use client certificates to authenticate the Web service requests. The client now must have not only a truststore (for authenticating the server) but also a keystore (for authenticating itself to the

server. The client provides an SSL certificate chain with a root signed by a trusted CA (the same root as in the previous case). This can be done by setting both key and trust stores for the SSL context of Web service requests. The former is used to provide credentials to the server, the latter is used to authenticate the server. In this case, the server uses a Certificates realm to authenticate the client, based on the signer of the client credentials and the trust anchors in the server (or, application server) truststore.

Note that the REST Web service is defined as a Web service in the server application, with security settings defined in web.xml. However instead of specifying form-based authentication, which you could have done in the previous assignment, you will instead be using basic or certificate-based authentication. To complete the assignment, you should enforce role-based access control for the Web service clients. The distinguished name in the client certificate will be used to identify clients for RBAC.

Follow these steps to set up your server and client keystore and truststore:

1. Generate the key pair and (self-signed) certificate for the root CA. Store this say in a keystore `caroot.jks` with key alias root-key. The intention is that this be stored offline.
2. Export the certificate as a PEM file, say `root-cert.pem`.
3. Create a truststore, say `cacerts.jks`, that contains this certificate as its only trust anchor, by importing the root CA certificate into an empty truststore.
4. Create the server key pair, with its own self-signed certificate, in what will become the server keystore, say `keystore.jks`, with key alias `s1as` (this alias is assumed by Glassfish).
5. Generate a CSR for the server's key pair , exporting this to a file `server-cert.csr`.
6. Using the root CA's private key (in `caroot.jks`), generate a server certificate from this CSR, writing it to a file, say `server-cert.cer`.
7. Import this certificate into the server keystore, `keystore.jks`, with key alias `s1as`.
8. Once you follow these steps, you will have generated the credentials for the server *administrator*, signed by your root CA, and stored in the server keystore. You will still need to generate the credentials for the server *instance*. Use the same server truststore and keystore files. Using the sequence described above, generate the server instance credentials signed by your root CA, and store them in the server keystore with key alias `glassfish-instance` (This alias is assumed by Glassfish).
9. The server keystore, `keystore.jks`, and truststore, `cacerts.jks`, should be saved in the config subdirectory of the domain directory in the application server installation. You should back up the existing keystore and truststore.

At this point you have set up a framework for a client to authenticate the server, by requiring that the server present an SSL certificate signed by the trusted root CA[1]. The truststore should be available to the client, so it can determine if it should trust the certificate provided by the server. The client can authenticate to the server by presenting its own credentials. If you are using BASIC authentication, then these credentials are a pair of a user name and password. If you are using client certificates for authentication, then the client credentials must be generated by the client and then signed by the root CA (in a similar manner as above), stored in a keystore at the client, and the server truststore must include the certificate of the root CA as a trust anchor[2].

To run the client, execute the following command from a shell (such as bash):

```
java –jar ChatClient.jar --truststore ... --passwordfile ... --uri ...
--sender ... --password ...
```

If you are using client certificates to authenticate, then you should type:

```
java –jar ChatClient.jar --keystore ... --truststore ... --passwordfile
... --uri ... --sender ... --certs
```

The command line arguments include (where appropriate) the locations of the client keystore and truststore, the passwords for the keystore and truststore, as well as for keys in the keystore, the URI for the server[3], the sender ID for posting messages, and the password if using BASIC authentication. The `--certs` option sets a flag that indicates you intend to authenticate with client certificates.

Submission

Your solution should be developed for Java 8 and Bouncycastle 1.54[4]. In addition, record short flash, mpeg, avi or Quicktime videos of a demonstration of your assignment working. Make sure that your name appears at the beginning of the video. *Do not provide private information such as your email or cwid in the video*.

Your solution should be uploaded via the Canvas classroom. Your solution should consist of a zip archive with one folder, identified by your name. Within that folder, you should have the Eclipse projects for the complete chat application, including the REST client. You should also provide a completed rubric for your assignment, as well as videos demonstrating the working of your assignment. Finally the root folder should contain the jar file for your compiled client, as well as the ear file for your

[1] Note that we are not asking you to do hostname verification of the server, because this would require that you control the DNS hostname of the server. Although this exposes your app to man-in-the-middle attacks, the risk is considerably less because you are only trusting certificates signed by your own CA.

[2] It goes without saying that the client and server should not use the same keystore and truststore!

[3] If running on the same machine as the client, this would be `https://localhost:8181/chat-rest`.

[4] You need BouncyCastle to sign the certificates, but since this is being done offline for this assignment, you don't need BouncyCastle on the client or server at runtime.

server application.

It is important that you provide a completed rubric that documents your submission, included as a PDF document in your submission root folder. As part of your submission, export your Eclipse projects to your file system, and then include these folders as part of your archive file.