SHRINE 3.2.1 Chapter 12.5 - Using Third-Party-Signed Certificate for Browser Encryption

> **Optional Step**
>
> This step is optional for remote sites.

By default, our recommendation for a typical remote site is to have it submit a Certificate Signing Request (CSR) to the certificate authority (CA) of the SHRINE network to which they are joining. The CA will in turn generate a new certificate for the downstream site, and we will return that certificate, the hub certificate, and the CA certificate of the tier back to the downstream site. The site will then import the certificates into their shrine keystore file, and configure their `shrine.conf` and `server.xml` to point to the alias entry in the keystore that corresponds to the site.

A typical keystore after importing all the certs in would look like this (assuming that the downstream site is called "`shrine.example.edu`"):

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 3 entries

shrine.example.edu, Oct 31, 2018, PrivateKeyEntry,
Certificate fingerprint (SHA1): 26:0B:FE:98:21:BA:C8:5A:A5:F5:35:79:8E:81:1A:E9:F4:3B:FF:56

shrine-act-test-ca, Jul 31, 2019, trustedCertEntry,
Certificate fingerprint (SHA1): 13:4D:B5:5C:E3:48:A0:7B:9B:20:22:8B:0B:C1:BE:DD:B9:E4:1B:AD

shrine-act-test.hms.harvard.edu, Jul 31, 2019, trustedCertEntry,
Certificate fingerprint (SHA1): 52:82:A0:6D:D1:48:B2:EA:BB:2C:58:BD:E5:C7:3B:21:75:2B:46:F6
```

`shrine.conf` contains a block like this:

```
keystore {
    file = "/opt/shrine/shrine.keystore"
    password = "xxxxxx"
    privateKeyAlias = "shrine.example.edu"
    keyStoreType = "JKS"
    caCertAliases = ["shrine-act-test-ca"]
  }
```

And `server.xml` will contain a block like this:

```
<Connector port="6443" protocol="org.apache.coyote.http11.Http11NioProtocol"
               maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
               clientAuth="false" sslProtocol="TLS"
               keystoreFile="/opt/shrine/shrine.keystore"
               keystorePass="xxxxxx"
               keyAlias="shrine.example.edu"/>
```

When using the above approach a web browser attempting to access a shrine host configured in this way will generate a warning. The browser will not trust the CA because none of the CAs are public. Consequently, any certificate that the CA signs is not trusted by the browser. While the browser can be configured to ignore the warning and the CA can be manually imported into the browser's trust store, some downstream sites may opt for a more elegant approach. Using a third-party-signed certificate eliminates the warnings from the web browser; the root and intermediate CAs are already trusted by the browser.

Using a Third-Party-Signed Certificate for Serving https Requests in Tomcat.

A third-party certificate does *not* replace the network CA-signed certificate; the network CA-signed certificate is still required for signing all application-specific messages. This wiki does not attempt to cover any vendor-specific processes or output files, because those can vary over time and across industry. It is up to each remote site that chooses this option to work with its vendor on any necessary technical details.

This guide also assumes the possibility that a site may initially opt for network CA-signed certificate, but later switch over to a third-party certificate. For all examples used in our illustration below this guide uses a fictitious remote site called `shrine.example.edu`.

The first step for the remote site is to generate a certificate signing request (CSR) using a private key, and to send that CSR to an SSL/TLS vendor such as InCommon and Symantec. This step can be performed using either `openssl` or `keytool`. The vendor will in turn generate a certificate for the requested fully-qualified domain name (FQDN), and it may provide additional certificates for its root and intermediate CAs. The remote site should work with the vendor to concatenate all certificates together into one file, so that it would be possible to trace the chain of trust from the endpoint certificate all the way back to the root CA.

The next step is to ensure that the private key that was used to generate the CSR is in a separate file. If a utility such as `openssl` was used to generate the CSR, then a file containing the private key should already be present and this step is unnecessary. If, however, the site used `keytool` to generate the CSR, then the following commands should be run to extract the private key from the keystore into a separate file. Execute the following commands to extract the private key into a file called `private_key.pem` (the command below assumes that the keystore is called `shrine.keystore`):

```
$ keytool -importkeystore -srckeystore shrine.keystore -srcstorepass <source_keystore_password> -srcalias
shrine.example.edu -destalias shrine.example.edu -destkeystore shrine.keystore.p12 -deststoretype PKCS12 -
deststorepass <destination_keystore_password>
$ openssl pkcs12 -in shrine.keystore.p12 -nodes -nocerts -out private_key.pem
```

This private key should be guarded carefully. Ideal places include an encrypted disk volume and non-persistent, RAM-based disk (such as `/dev/shm` in CentOS or Debian). The key can also be stored in an offline and physically secure location.

Next, upon receiving the signed certificate from the third-party CA, the remote site should then bundle the private key with the chained certificates into a PKCS12 file (`.pfx` or `.p12` suffix) so that SHRINE can verify the chain of trust from the endpoint certificate all the way back to the root CA. Without this trust verification, SHRINE will be unable to use the certificates as intended. For our purpose, the PKCS#12 file format is ideal because it can combine a private key with all corresponding chained certificates into a single entry in the file, and because the format is accessible by both `openssl` and `keytool`. In the following command (again, assuming that the private key file is called `private_key.pem`), the `certificates_file` contains the new endpoint certificate plus all intermediate certificates plus the root certificate, and the `ca_file` contains only the intermediate certificates plus the root certificate. Run this command to bundle the private key, the endpoint certificate, and all intermediate and root certificates into one single entry within the PKCS12 store:

```
$ openssl pkcs12 -export -in <certificates_file> -inkey private_key.pem -out shrine.keystore.p12.temp -name
shrine.example.edu-https -CAfile <ca_file> -chain -password pass:<your_pkcs12_file_password>
```

Note here that the value of the `-name` field is "shrine.example.edu-https". The reason is that we will eventually be importing this entire entry back into the main Java keystore where another entry called `shrine.example.edu` already exists. Recalling our earlier discussion about the purposes of certificates, we note that `shrine.example.edu` entry is responsible for application message signing, while `shrine.example.edu-https` entry is responsible for web browser encryption. Therefore two distinct keystore entries are necessary.

The newly constructed PKCS12 file should now be imported into the original `shrine.keystore`. Execute the following command, and if prompted to overwrite existing entry, answer "yes":

```
$ keytool -importkeystore -srckeystore shrine.keystore.p12.temp -srcstoretype pkcs12 -srcstorepass
<your_pkcs12_file_password> -srcalias shrine.example.edu-https -destkeystore shrine.keystore -deststoretype jks
-deststorepass <your_destination_keystore_password> -destalias shrine.example.edu-https
```

Once the import is completed, your shrine.keystore should now look like this:

```
Keystore type: jks
Keystore provider: SUN

Your keystore contains 4 entries
shrine.example.edu, Oct 31, 2018, PrivateKeyEntry,
Certificate fingerprint (SHA1): 26:0B:FE:98:21:BA:C8:5A:A5:F5:35:79:8E:81:1A:E9:F4:3B:FF:56

shrine.example.edu-https, Jul 31, 2019, PrivateKeyEntry,
Certificate fingerprint (SHA1): EF:AF:5E:D3:1A:97:AA:F2:6D:50:B8:9A:23:98:B5:2C:0C:18:C3:4B

shrine-act-test-ca, Jul 31, 2019, trustedCertEntry,
Certificate fingerprint (SHA1): 13:4D:B5:5C:E3:48:A0:7B:9B:20:22:8B:0B:C1:BE:DD:B9:E4:1B:AD

shrine-act-test.hms.harvard.edu, Jul 31, 2019, trustedCertEntry,
Certificate fingerprint (SHA1): 52:82:A0:6D:D1:48:B2:EA:BB:2C:58:BD:E5:C7:3B:21:75:2B:46:F6
```

Notice that there are now two (2) entries associated with shrine.example.edu: the original one continues to be used for application signing, and we will configure the "`-https`" entry for use by Tomcat to encrypt web browser traffic.

Next, configure the alias entry in Tomcat's `server.xml` to point to the newly imported, third-party-generated certificate:

```
<Connector port="6443" protocol="org.apache.coyote.http11.Http11NioProtocol"
               maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
               clientAuth="false" sslProtocol="TLS"
               keystoreFile="/opt/shrine/shrine.keystore"
               keystorePass="xxxxxxxxxx"
               keyAlias="shrine.example.edu-https"/>
```

Once all configuration files have been updated, restart Tomcat. If everything has been configured properly, you should no longer see any warning messages from your browser regarding untrusted certificates when accessing your SHRINE host.