Version: 9.4.19.v20190610

Configuring SSL/TLS

This document provides an overview of how to configure SSL and TLS for Jetty.

Configuring Jetty for SSL

To configure Jetty for SSL, complete the tasks in the following sections:

- Generating Key Pairs and Certificates
- Requesting a Trusted Certificate
- Loading Keys and Certificates
- Configuring the Jetty SslContextFactory

TLS and SSL versions

Which browser/OS supports which protocols can be found on Wikipedia.

- TLS v1.2: The protocol which should be used wherever possible. All CBC based ciphers are supported since Java 7, the new GCM modes are supported since Java 8.

Older Protocols

TLS v1.0, v1.1 and SSL v3 are no longer supported by default. If your Jetty implementation requires these protocols for legacy support, they can be enabled manually.

> **✱ Note**
>
> Once TLS v1.3 is released, there will be no workaround available for TLS v1.0 or v1.1. Plans for TLS v1.3 include banning ciphers with known vulnerabilities from being present at any level. It is recommended to upgrade any clients using these ciphers as soon as possible or face being locked into a outdated version of Jetty, Java or even OS.

By default, Jetty excludes these ciphers in the `SslContextFactory`. You can re-enable these by re-declaring the ciphers you want excluded in code:

```
SslContextFactory.Server sslContextFactory = new SslContextFactory.Server();
sslContextFactory.setExcludeCipherSuites("^.*_(MD5|SHA|SHA1)$");
```

If, after making these changes, you still have issues using these ciphers they are likely being blocked at the JVM level. Locate the `$JAVA_HOME/jre/lib/security/` directory for the `java.security` file and examine it for any configuration that is excluding *ciphers* or *algorithms* (depending on the version of the JVM you are using the nomenclature may be different).

Understanding Certificates and Keys

Configuring SSL can be a confusing experience of keys, certificates, protocols and formats, thus it helps to have a reasonable understanding of the basics. The following links provide some good starting points:

- Certificates:
  - SSL Certificates HOWTO
  - Mindprod Java Glossary: Certificates
- Keytool:
  - Keytool for Unix

- Keytool for Windows

- Other tools:
  - IBM Keyman

- OpenSSL:
  - OpenSSL FAQ

OpenSSL vs. Keytool

For testing, the `keytool` utility bundled with the JDK provides the simplest way to generate the key and certificate you need.

You can also use the OpenSSL tools to generate keys and certificates, or to convert those that you have used with Apache or other servers. Since Apache and other servers commonly use the OpenSSL tool suite to generate and manipulate keys and certificates, you might already have some keys and certificates created by OpenSSL, or you might also prefer the formats OpenSSL produces.

If you want the option of using the same certificate with Jetty or a web server such as Apache not written in Java, you might prefer to generate your private key and certificate with OpenSSL.

Generating Key Pairs and Certificates

The simplest way to generate keys and certificates is to use the `keytool` application that comes with the JDK, as it generates keys and certificates directly into the keystore. See Generating Keys and Certificates with JDK's keytool.

If you already have keys and certificates, see Loading Keys and Certificates to load them into a JSSE keystore. This section also applies if you have a renewal certificate to replace one that is expiring.

The examples below generate only basic keys and certificates. You should read the full manuals of the tools you are using if you want to specify:

- The key size
- The certificate expiration date
- Alternate security providers

Generating Keys and Certificates with JDK's keytool

The following command generates a key pair and certificate directly into file `keystore`:

```
$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA
```

> **✱ Note**
>
> The DSA key algorithm certificate produces an error after loading several pages. In a browser, it displays a message "Could not establish an encrypted connection because certificate presented by localhost as an invalid signature." The solution is to use RSA for the key algorithm.

This command prompts for information about the certificate and for passwords to protect both the keystore and the keys within it. The only mandatory response is to provide the fully qualified host name of the server at the "first and last name" prompt. For example:

```
$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA -sigalg SHA256withRSA
 Enter keystore password:  password
 What is your first and last name?
   [Unknown]:  jetty.eclipse.org
 What is the name of your organizational unit?
   [Unknown]:  Jetty
 What is the name of your organization?
   [Unknown]:  Mort Bay Consulting Pty. Ltd.
 What is the name of your City or Locality?
   [Unknown]:
 What is the name of your State or Province?
   [Unknown]:
 What is the two-letter country code for this unit?
   [Unknown]:
 Is CN=jetty.eclipse.org, OU=Jetty, O=Mort Bay Consulting Pty. Ltd.,
 L=Unknown, ST=Unknown, C=Unknown correct?
   [no]:  yes

 Enter key password for <jetty>
         (RETURN if same as keystore password):
 $
```

You now have the minimal requirements to run an SSL connection and could proceed directly to configure an SSL connector. However, the browser *will not* trust the certificate you have generated, and prompts the user to this effect. While what you have at this point is often sufficient for testing, most public sites need a trusted certificate, which is demonstrated in the section generating a CSR with keytool.

If you want to use only a self signed certificate for some kind of internal admin panel add -validity <days> to the keytool call above, otherwise your certificate is only valid for one month.

If you are using Java 8 or later, then you may also use the SAN extension to set one or more names that the certificate applies to:

```
$ keytool -keystore keystore -alias jetty -genkey -keyalg RSA -sigalg SHA256withRSA -ext
'SAN=dns:jetty.eclipse.org,dns:*.jetty.org'
 ...
```

Generating Keys and Certificates with OpenSSL

The following command generates a key pair in the file `jetty.key`:

```
$ openssl genrsa -aes128 -out jetty.key
```

You might also want to use the -`rand` file argument to provide an arbitrary file that helps seed the random number generator.

The following command generates a certificate for the key into the file `jetty.crt`:

```
$ openssl req -new -x509 -newkey rsa:2048 -sha256 -key jetty.key -out jetty.crt
```

Adding -sha256 ensures to get a certificate with the now recommended SHA-256 signature algorithm. For the those with heightened security in mind, add -b4096 to get a 4069 bit key.

The next command prompts for information about the certificate and for passwords to protect both the keystore and the keys within it. The only mandatory response is to provide the fully qualified host name of the server at the "Common Name" prompt. For example:

```
$ openssl genrsa -aes128 -out jetty.key
Generating RSA private key, 2048 bit long modulus
..............+++
.....................................................................+++
e is 65537 (0x10001)
Enter pass phrase for jetty.key:
Verifying - Enter pass phrase for jetty.key:

$ openssl req -new -x509 -newkey rsa:2048 -sha256 -key jetty.key -out jetty.crt
Enter pass phrase for jetty.key:
You are about to be asked to enter information that will be incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank.
For some fields there will be a default value.
If you enter '.', the field will be left blank.

Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Mort Bay Consulting Pty. Ltd.
Organizational Unit Name (eg, section) []:Jetty
Common Name (e.g. server FQDN or YOUR name) []:jetty.eclipse.org
Email Address []:

$
```

You now have the minimal requirements to run an SSL connection and could proceed directly to] to load these keys and certificates into a JSSE keystore. However the browser *will not* trust the certificate you have generated, and prompts the user to this effect. While what you have at this point is often sufficient for testing, most public sites need a trusted certificate, which is demonstrated in the section, xref:generating-csr-from-openssl[to obtain a certificate.

Using Keys and Certificates from Other Sources

If you have keys and certificates from other sources, you can proceed directly to Loading Keys and Certificates.

Requesting a Trusted Certificate

The keys and certificates generated with JDK's `keytool` and OpenSSL are sufficient to run an SSL connector. However the browser will not trust the certificate you have generated, and it will prompt the user to this effect.

To obtain a certificate that most common browsers will trust, you need to request a well-known certificate authority (CA) to sign your key/certificate. Such trusted CAs include: AddTrust, Entrust, GeoTrust, RSA Data Security, Thawte, VISA, ValiCert, Verisign, and beTRUSTed, among others. Each CA has its own instructions (look for JSSE or OpenSSL sections), but all involve a step that generates a certificate signing request (CSR).

Generating a CSR with keytool

The following command generates the file `jetty.csr` using `keytool` for a key/cert already in the keystore:

```
$ keytool -certreq -alias jetty -keystore keystore -file jetty.csr
```

Generating a CSR from OpenSSL

The following command generates the file `jetty.csr` using OpenSSL for a key in the file `jetty.key`:

```
$ openssl req -new -key jetty.key -out jetty.csr
```

Notice that this command uses only the existing key from `jetty.key` file, and not a certificate in `jetty.crt` as generated with OpenSSL. You need to enter the details for the certificate again.

Loading Keys and Certificates

Once a CA has sent you a certificate, or if you generated your own certificate without `keytool`, you need to load it into a JSSE keystore.

> **✱ Note**
>
> You need both the private key and the certificate in the JSSE keystore. You should load the certificate into the keystore used to generate the CSR with `keytool`. If your key pair is not already in a keystore (for example, because it has been generated with OpenSSL), you need to use the PKCS12 format to load both key and certificate (see PKCKS12 Keys &Certificates).

Loading Certificates with keytool

You can use `keytool` to load a certificate in PEM form directly into a keystore. The PEM format is a text encoding of certificates; it is produced by OpenSSL, and is returned by some CAs. An example PEM file is:

```
jetty.crt
-----BEGIN CERTIFICATE-----
MIICSDCCAfKgAwIBAgIBADANBgkqhkiG9w0BAQQFADBUMSYwJAYDVQQKEx1Nb3J0
IEJheSBDb25zdWx0aW5nIFB0eS4gTHRkLjEOMAwGA1UECxMFSmV0dHkxGjAYBgNV
BAMTEWpldHR5Lm1vcnRiYXkub3JnMB4XDTAzMDQwNjEzMTk1MFoXDTAzMDUwNjEz
MTk1MFowVDEmMCQGA1UEChMdTW9ydCBCYXkgQ29uc3VsdGluZyBQdHkuIEx0ZC4x
DjAMBgNVBAsTBUpldHR5MRowGAYDVQQDExFqZXR0eS5tb3J0YmF5Lm9yZzBcMA0G
CSqGSIb3DQEBAQUAA0sAMEgCQQC5V4oZeVdhdhHqa9L2/ZnKySPWUqqy81riNfAJ
7uALW0kEv/LtlG34dOOcVVt/PK8/bU4dlolnJx1SpiMZbKsFAgMBAAGjga4wgasw
HQYDVR0OBBYEFFV1gbB1XRvUx1UofmifQJS/MCYwMHwGA1UdIwR1MHOAFFV1gbB1
XRvUx1UofmifQJS/MCYwoVikVjBUMSYwJAYDVQQKEx1Nb3J0IEJheSBDb25zdWx0
aW5nIFB0eS4gTHRkLjEOMAwGA1UECxMFSmV0dHkxGjAYBgNVBAMTEWpldHR5Lm1v
cnRiYXkub3JnggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEEBQADQQA6NkaV
OtXzP4ayzBcgK/qSCmF44jdcARmrXhiXUcXzjxsLjSJeYPJojhUdC2LQKy+p4ki8
Rcz6oCRvCGCe5kDB
-----END CERTIFICATE-----
```

The following command loads a PEM encoded certificate in the `jetty.crt` file into a JSSE keystore:

```
$ keytool -keystore keystore -import -alias jetty -file jetty.crt -trustcacerts
```

If the certificate you receive from the CA is not in a format that `keytool` understands, you can use the `openssl` command to convert formats:

```
$ openssl x509 -in jetty.der -inform DER -outform PEM -out jetty.crt
```

Loading Keys and Certificates via PKCS12

If you have a key and certificate in separate files, you need to combine them into a PKCS12 format file to load into a new keystore. The certificate can be one you generated yourself or one returned from a CA in response to your CSR.

The following OpenSSL command combines the keys in `jetty.key` and the certificate in the `jetty.crt` file into the `jetty.pkcs12` file:

```
$ openssl pkcs12 -inkey jetty.key -in jetty.crt -export -out jetty.pkcs12
```

If you have a chain of certificates, because your CA is an intermediary, build the PKCS12 file as follows:

```
$ cat example.crt intermediate.crt [intermediate2.crt] ... rootCA.crt > cert-chain.txt
$ openssl pkcs12 -export -inkey example.key -in cert-chain.txt -out example.pkcs12
```

> **✱ Note**
>
> The order of certificates must be from server to rootCA, as per RFC2246 section 7.4.2.

OpenSSL asks for an *export password*. A non-empty password is required to make the next step work. Load the resulting PKCS12 file into a JSSE keystore with `keytool`:

```
$ keytool -importkeystore -srckeystore jetty.pkcs12 -srcstoretype PKCS12 -destkeystore keystore
```

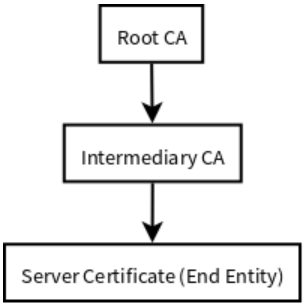
Renewing Certificates

If you are updating your configuration to use a newer certificate, as when the old one is expiring, just load the newer certificate as described in the section, Loading Keys and Certificates. If you imported the key and certificate originally using the PKCS12 method, use an alias of "1" rather than "jetty", because that is the alias the PKCS12 process enters into the keystore.

Layout of keystore and truststore

The `keystore` only contains the server's private key and certificate.

Figure 6.1. Certificate chain



The structure of KeyStore file:

```
├── PrivateKeyEntry
│   ├── PrivateKey
│   ├── Certificate chain
│   │   ├── Server certificate (end entity)
│   │   ├── Intermediary CA certificate
│   │   └── Root CA certificate
├── TrustedCertEntry
│   └── Intermediary CA certificate
└── TrustedCertEntry
    └── Root CA certificate
```

> **✱ Note**
>
> Both the `Intermediary CA certificate` and `Root CA certificate` are optional.

```
$ cd $JETTY_BASE
$ keytool -list -keystore etc/keystore -storetype jks -storepass '' -v

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 3 entries

Alias name: *.example.com
Creation date: Sep 20, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 3
Certificate[1]:
Owner: CN=*.example.com, OU=Web Servers, O="Example.com Co.,Ltd.", C=CN
Issuer: CN="Example.com Co.,Ltd. ETP CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Serial number: b63af619ff0b4c368735113ba5db8997
Valid from: Mon Sep 12 15:09:49 CST 2016 until: Wed Sep 12 15:09:49 CST 2018
Certificate fingerprints:
         MD5:  D9:26:CC:27:77:9D:26:FE:67:4C:BE:FF:E3:95:1E:97
```

```
         SHA1: AF:DC:D2:65:6A:33:42:E3:81:9E:4D:19:0D:22:20:C7:6F:2F:11:D0
         SHA256: 43:E8:21:5D:C6:FB:A0:7D:5D:7B:9C:8B:8D:E9:4B:52:BF:50:0D:90:4F:61:C2:18:9E:89:AA:4C:C2:93:BD:32
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 44 9B AD 31 E7 FE CA D5   5A 8E 17 55 F9 F0 1D 6B  D..1....Z..U...k
0010: F5 A5 8F C1                                        ....
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:false
  PathLen: undefined
]

#3: ObjectId: 2.5.29.37 Criticality=true
ExtendedKeyUsages [
  serverAuth
  clientAuth
]

#4: ObjectId: 2.5.29.15 Criticality=true
KeyUsage [
  DigitalSignature
  Key_Encipherment
  Data_Encipherment
]

#5: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 7D 26 36 73 61 5E 08 94   AD 25 13 46 DB DB 95 25  .&6sa^...%.F...%
0010: BF 82 5A CA                                        ..Z.
]
]

Certificate[2]:
Owner: CN="Example.com Co.,Ltd. ETP CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Issuer: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Serial number: f6e7b86f6fdb467f9498fb599310198f
Valid from: Wed Nov 18 00:00:00 CST 2015 until: Sun Nov 18 00:00:00 CST 2035
Certificate fingerprints:
         MD5:  ED:A3:91:57:D8:B8:6E:B1:01:58:55:5C:33:14:F5:99
         SHA1: D9:A4:93:9D:A6:F8:A3:F9:FD:85:51:E2:C5:2E:0B:EE:80:E7:D0:22
         SHA256: BF:54:7A:F6:CA:0C:FA:EF:93:B6:6B:6E:2E:D7:44:A8:40:00:EC:69:3A:2C:CC:9A:F7:FE:8E:6F:C0:FA:22:38
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.15 Criticality=true
KeyUsage [
  Key_CertSign
  Crl_Sign
]

#4: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 44 9B AD 31 E7 FE CA D5   5A 8E 17 55 F9 F0 1D 6B  D..1....Z..U...k
0010: F5 A5 8F C1                                        ....
]
]

Certificate[3]:
Owner: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Issuer: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Serial number: f0a45bc9972c458cbeae3f723055f1ac
Valid from: Wed Nov 18 00:00:00 CST 2015 until: Sun Nov 18 00:00:00 CST 2114
Certificate fingerprints:
         MD5:  50:61:62:22:71:60:F7:69:2E:27:42:6B:62:31:82:79
         SHA1: 7A:6D:A6:48:B1:43:03:3B:EA:A0:29:2F:19:65:9C:9B:0E:B1:03:1A
         SHA256: 05:3B:9C:5B:8E:18:61:61:D1:9C:AA:0E:8C:B1:EA:44:C2:6E:67:5D:96:30:EC:8C:F6:6F:E1:EC:AD:00:60:F1
```

```
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.15 Criticality=true
KeyUsage [
  Key_CertSign
  Crl_Sign
]

#4: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]



*******************************************
*******************************************


Alias name: example.com co.,ltd. etp ca
Creation date: Sep 20, 2016
Entry type: trustedCertEntry

Owner: CN="Example.com Co.,Ltd. ETP CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Issuer: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Serial number: f6e7b86f6fdb467f9498fb599310198f
Valid from: Wed Nov 18 00:00:00 CST 2015 until: Sun Nov 18 00:00:00 CST 2035
Certificate fingerprints:
         MD5:  ED:A3:91:57:D8:B8:6E:B1:01:58:55:5C:33:14:F5:99
         SHA1: D9:A4:93:9D:A6:F8:A3:F9:FD:85:51:E2:C5:2E:0B:EE:80:E7:D0:22
         SHA256: BF:54:7A:F6:CA:0C:FA:EF:93:B6:6B:6E:2E:D7:44:A8:40:00:EC:69:3A:2C:CC:9A:F7:FE:8E:6F:C0:FA:22:38
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.15 Criticality=true
KeyUsage [
  Key_CertSign
  Crl_Sign
]

#4: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 44 9B AD 31 E7 FE CA D5   5A 8E 17 55 F9 F0 1D 6B  D..1....Z..U...k
0010: F5 A5 8F C1                                        ....
]
]



*******************************************
*******************************************


Alias name: example.com co.,ltd. root ca
Creation date: Sep 20, 2016
Entry type: trustedCertEntry
```

```
Owner: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Issuer: CN="Example.com Co.,Ltd. Root CA", OU=CA Center, O="Example.com Co.,Ltd.", C=CN
Serial number: f0a45bc9972c458cbeae3f723055f1ac
Valid from: Wed Nov 18 00:00:00 CST 2015 until: Sun Nov 18 00:00:00 CST 2114
Certificate fingerprints:
         MD5:  50:61:62:22:71:60:F7:69:2E:27:42:6B:62:31:82:79
         SHA1: 7A:6D:A6:48:B1:43:03:3B:EA:A0:29:2F:19:65:9C:9B:0E:B1:03:1A
         SHA256: 05:3B:9C:5B:8E:18:61:61:D1:9C:AA:0E:8C:B1:EA:44:C2:6E:67:5D:96:30:EC:8C:F6:6F:E1:EC:AD:00:60:F1
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]

#2: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

#3: ObjectId: 2.5.29.15 Criticality=true
KeyUsage [
  Key_CertSign
  Crl_Sign
]

#4: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: A6 BD 5F B3 E8 7D 74 3D   20 44 66 1A 16 3B 1B DF  .._...t= Df..;..
0010: E6 E6 04 46                                        ...F
]
]



*******************************************
*******************************************
```

In addition, you can split `$JETTY/etc/keystore` as two files. One is `$JETTY/etc/keystore` which only contains the server's private key and certificate, the other is `$JETTY/etc/truststore` which contains intermediary CA and root CA.

The structure of `$JETTY/etc/keystore`.

```
└── PrivateKeyEntry
    ├── PrivateKey
    └── Certificate chain
        └── Server certificate (end entity)
```

The structure of `$JETTY/etc/truststore`.

```
├── TrustedCertEntry
│   └── Intermediary CA certificate
└── TrustedCertEntry
    └── Root CA certificate
```

Configuring the Jetty SslContextFactory

The generated SSL certificates from above are held in the key store are configured in an instance of SslContextFactory.Server object.

The `SslContextFactory` is responsible for:

- Creating the Java `SslEngine` used by Jetty's Connectors and Jetty's Clients (HTTP/1, HTTP/2, and WebSocket).
- Managing Keystore Access
- Managing Truststore Access
- Managing Protocol selection via Excludes / Includes list
- Managing Cipher Suite selection via Excludes / Includes list
- Managing order of Ciphers offered (important for TLS/1.2 and HTTP/2 support)
- SSL Session Caching options
- Certificate Revocation Lists and Distribution Points (CRLDP)
- OCSP Support
- Client Authentication Support

For Jetty Connectors, the configured `SslContextFactory.Server` is injected into a specific ServerConnector `SslConnectionFactory`.

For Jetty Clients, the various constructors support using a configured `SslContextFactory.Client`.

While the `SslContextFactory` can operate without a keystore (this mode is most suitable for the various Jetty Clients) it is best practice to at least configure the keystore being used.

setKeyStorePath
The configured keystore to use for all SSL/TLS in configured Jetty Connector (or Client).

> **✱ Note**
>
> As a keystore is vital security information, it can be desirable to locate the file in a directory with **very** restricted access.

setKeyStorePassword
The keystore password may be set here in plain text, or as some measure of protection from casual observation, it may be obfuscated using the Password class.
setTrustStorePath
This is used if validating client certificates and is typically set to the same path as the keystore.
setKeyManagerPassword
The password that is passed to the `KeyManagerFactory.init(...)`. If there is no `keymanagerpassword`, then the `keystorepassword` is used instead. If there is no `trustmanager` set, then the keystore is used as the trust store and the `keystorepassword` is used as the truststore password.
setExcludeCipherSuites / setIncludeCipherSuites
This allows for the customization of the selected Cipher Suites that will be used by SSL/TLS.
setExcludeProtocols / setIncludeProtocols
This allows for the customization of the selected Protocols that will be used by SSL/TLS.

> **✱ Note**
>
> When working with Includes / Excludes, it is important to know that **Excludes will always win.** The selection process is to process the JVM list of available Cipher Suites or Protocols against the include list, then remove the excluded ones. Be aware that each Include / Exclude list has a Set method (replace the list) or Add method (append the list).

> **! Caution**
>
> The keystore and truststore passwords may also be set using the system properties: `org.eclipse.jetty.ssl.keypassword` `org.eclipse.jetty.ssl.password`. This is *not* a recommended usage.

Conscrypt SSL

Jetty includes support for Google's Conscrypt SSL, which is built on their fork of OpenSSL, BoringSSL. Implementing Conscrypt for the server or client is very straightforward process - simply instantiate an instance of Conscrypt's `OpenSSLProvider` and set `Conscrypt` as a provider for Jetty's `SslContextFactory`:

```
...
Security.addProvider(new OpenSSLProvider());
...
SslContextFactory.Server sslContextFactory = new SslContextFactory.Server();
sslContextFactory.setKeyStorePath("path/to/keystore");
sslContextFactory.setKeyStorePassword("CleverKeyStorePassword");
sslContextFactory.setKeyManagerPassword("OBF:VerySecretManagerPassword");
sslContextFactory.setProvider("Conscrypt");
...
```

If you are using the Jetty Distribution, please see the section on enabling the Conscrypt SSL module.

If you are using Conscrypt with Java 8, you must exclude `TLSv1.3` protocol as it is now enabled per default with Conscrypt 2.0.0 but not supported by Java 8.

Configuring SNI

From Java 8, the JVM contains support for the Server Name Indicator (SNI) extension, which allows a SSL connection handshake to indicate one or more DNS names that it applies to.

To support this, the `SslContextFactory` is used. The `SslContextFactory` will look for multiple X509 certificates within the keystore, each of which may have multiple DNS names (including wildcards) associated with the Subject Alternate Name extension. When using the `SslContextFactory`, the correct certificate is automatically selected if the SNI extension is present in the handshake.

Disabling/Enabling Specific Cipher Suites

New cipher suites are always being developed to stay ahead of attacks. It's only a matter of time before the best of suites is exploited though, and making sure your server is up-to-date in this regard is paramount for any implementation. As an example, to avoid the BEAST attack it is necessary to configure a specific set of cipher suites. This can either be done via SslContext.setIncludeCipherSuites(java.lang.String…) or viaSslContext.setExcludeCipherSuites(java.lang.String…).

It's crucial that you use the *exact* names of the cipher suites as used/known by the JDK. You can get them by obtaining an instance of SSLEngine and call `getSupportedCipherSuites()`. Tools like ssllabs.com might report slightly different names which will be ignored.

> **Important**
>
> It is important to stay up-to-date with the latest supported cipher suites. Be sure to consult Oracle's JRE and JDK Cryptographic Roadmap frequently for recent and upcoming changes to supported ciphers.

> **Important**
>
> It's recommended to install the Java Cryptography Extension (JCE) Unlimited Strength policy files in your JRE to get full strength ciphers such as AES-256. The files can be found on the Java download page. Just overwrite the two present JAR files in `<JRE_HOME>/lib/security/`.

Both `setIncludeCipherSuites` and `setExcludeCipherSuites` can be fed by the exact cipher suite name used in the JDK or by using regular expressions. If you have a need to adjust the Includes or Excludes, then this is best done with a custom XML that configures the `SslContextFactory` to suit your needs.

> **Note**
>
> Jetty **does** allow users to enable weak/deprecated cipher suites (or even no cipher suites at all). By default, if you have these suites enabled warning messages will appear in the server logs.

To do this, first create a new `${jetty.base}/etc/tweak-ssl.xml` file (this can be any name, just avoid prefixing it with "jetty-").

```
<!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN"
          "http://www.eclipse.org/jetty/configure_9_3.dtd">
<!-- Tweak SsslContextFactory Includes / Excludes -->
<Configure id="sslContextFactory" class="org.eclipse.jetty.util.ssl.SslContextFactory$Server">
  <!-- Mitigate SLOTH Attack -->
  <Call name="addExcludeCipherSuites">
    <Arg>
      <Array type="String">
        <Item>.*_RSA_.*SHA1$</Item>
        <Item>.*_RSA_.*SHA$</Item>
        <Item>.*_RSA_.*MD5$</Item>
      </Array>
    </Arg>
  </Call>
</Configure>
```

This new XML will configure the id `sslContextFactory` further (this id is first created by the `ssl` module and its associated `${jetty.home}/etc/jetty-ssl-context.xml`). You can do anything you want with the `SslContextFactory` in use by the Jetty Distribution from this tweaked XML.

To make sure that your `${jetty.base}` uses this new XML, add it to the end of your `${jetty.base}/start.ini` or `${jetty.base}/start.d/server.ini`.

```
$ cd /path/to/mybase
$ ls -l
drwxrwxr-x.  2 user group  4096 Feb  2 11:47 etc/
-rw-rw-r--.  1 user group  4259 Feb  2 11:47 start.ini
$ tail start.ini
# Module: https
--module=https
etc/tweak-ssl.xml
$
```

> **Note**
>
> The default `SslContextFactory` implementation applies the latest SSL/TLS recommendations surrounding vulnerabilities in SSL/TLS. Check the release notes (the `VERSION.txt` found in the root of the Jetty Distribution, or the alternate (classified *version*) artifacts for the `jetty-project` component on Maven Central) for updates. The Java JVM also applies exclusions at the JVM level and, as such, if you have a need to enable something that is generally accepted by the industry as being insecure or vulnerable you will likely have to enable it in **both** the Java JVM and your Jetty configuration.

> **Tip**
>
> You can enable the `org.eclipse.jetty.util.ssl` named logger at DEBUG level to see what the list of selected Protocols and Cipher suites are at startup of Jetty.

Additional Include / Exclude examples:

Example: Include all ciphers which support Forward Secrecy using regex:

```
<!-- Enable Forward Secrecy Ciphers.
     Note: this replaces the default Include Cipher list -->
<Set name="IncludeCipherSuites">
  <Array type="String">
    <Item>TLS_DHE_RSA.*</Item>
    <Item>TLS_ECDHE.*</Item>
  </Array>
</Set>
```

Example: Exclude all old, insecure or anonymous cipher suites:

```
<!-- Eliminate Old / Insecure / Anonymous Ciphers -->
<Call name="addExcludeCipherSuites">
  <Arg>
    <Array type="String">
      <Item>.*NULL.*</Item>
      <Item>.*RC4.*</Item>
      <Item>.*MD5.*</Item>
      <Item>.*DES.*</Item>
      <Item>.*DSS.*</Item>
    </Array>
  </Arg>
</Call>
```

Example: Since 2014 SSLv3 is considered insecure and should be disabled.

```
<!-- Eliminate Insecure Protocols -->
<Call name="addExcludeProtocols">
  <Arg>
   <Array type="java.lang.String">
     <Item>SSL</Item>
     <Item>SSLv2</Item>
     <Item>SSLv2Hello</Item>
     <Item>SSLv3</Item>
   </Array>
  </Arg>
</Call>
```

> **✱ Note**
>
> Note that disabling SSLv3 prevents very old browsers like Internet Explorer 6 on Windows XP from connecting.

Example: TLS renegotiation could be disabled too to prevent an attack based on this feature.

```
<Set name="renegotiationAllowed">FALSE</Set>
```

You can view what cipher suites are enabled and disabled by performing a server dump.

To perform a server dump upon server startup, add `jetty.server.dumpAfterStart=true` to the command line when starting the server. You can also dump the server when shutting down the server instance by adding `jetty.server.dumpBeforeStop`.

Specifically, you will want to look for the `SslConnectionFactory` portion of the dump.

```
[my-base]$ java -jar ${JETTY_HOME}/start.jar jetty.server.dumpAfterStart=true

...
|   += SslConnectionFactory@18be83e4{SSL->http/1.1} - STARTED
|   |   += SslContextFactory@42530531(null,null) trustAll=false
|   |       +- Protocol Selections
|   |       |   +- Enabled (size=3)
|   |       |   |   +- TLSv1
|   |       |   |   +- TLSv1.1
|   |       |   |   +- TLSv1.2
|   |       |   +- Disabled (size=2)
|   |       |       +- SSLv2Hello - ConfigExcluded:'SSLv2Hello'
|   |       |       +- SSLv3 - JreDisabled:java.security, ConfigExcluded:'SSLv3'
|   |       +- Cipher Suite Selections
|   |           +- Enabled (size=15)
|   |           |   +- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
|   |           |   +- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
|   |           |   +- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
|   |           |   +- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
|   |           |   +- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
|   |           |   +- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
```

```
|   |           |   +- TLS_EMPTY_RENEGOTIATION_INFO_SCSV
|   |           |   +- TLS_RSA_WITH_AES_128_CBC_SHA256
|   |           |   +- TLS_RSA_WITH_AES_128_GCM_SHA256
|   |           +- Disabled (size=42)
|   |               +- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DHE_DSS_WITH_DES_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DHE_RSA_WITH_DES_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_DH_anon_WITH_DES_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_RSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_RSA_WITH_DES_CBC_SHA - JreDisabled:java.security, ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_RSA_WITH_NULL_MD5 - JreDisabled:java.security, ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- SSL_RSA_WITH_NULL_SHA - JreDisabled:java.security, ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_DHE_DSS_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_DHE_RSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_DH_anon_WITH_AES_128_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_DH_anon_WITH_AES_128_CBC_SHA256 - JreDisabled:java.security
|   |               +- TLS_DH_anon_WITH_AES_128_GCM_SHA256 - JreDisabled:java.security
|   |               +- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDHE_ECDSA_WITH_NULL_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDHE_RSA_WITH_NULL_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_ECDSA_WITH_NULL_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_RSA_WITH_NULL_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_anon_WITH_AES_128_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_ECDH_anon_WITH_NULL_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5 - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_WITH_3DES_EDE_CBC_MD5 - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_WITH_3DES_EDE_CBC_SHA - JreDisabled:java.security,
ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_WITH_DES_CBC_MD5 - JreDisabled:java.security, ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_KRB5_WITH_DES_CBC_SHA - JreDisabled:java.security, ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_RSA_WITH_AES_128_CBC_SHA - ConfigExcluded:'^.*_(MD5|SHA|SHA1)$'
|   |               +- TLS_RSA_WITH_NULL_SHA256 - JreDisabled:java.security
...
```

In the example above you can see both the enabled/disabled protocols and included/excluded cipher suites. For disabled or excluded protocols and ciphers, the reason they are disabled is given - either due to JVM restrictions, configuration or both. As a reminder, when configuring your includes/excludes, **excludes always win**.

Dumps can be configured as part of the `jetty.xml` configuration for your server. Please see the documentation on the Jetty Dump Tool for more information.

Previous
Chapter 6. Configuring Jetty Connectors
Top
Home
Next
SSL in the Jetty Distribution