
Configuration of WS-Security and HTTPS in OMI Installations

Revision: 0.1

Author: Bryan Carpenter

Open Middleware Infrastructure Institute
Room 6005, Building 21 (Faraday)
Highfield Campus
University of Southampton
Hampshire, SO17 1BJ
Tel: +44 (0)2380 58788

Revision No.	Author	Changes Made	Date Revised
0.1	Bryan Carpenter	Document created	18/01/07

Table of Contents

1.	<i>Introduction</i>	4
2.	<i>X509 Certificates and Keystores in OMII Installations</i>	5
3.	<i>Configuring OMII Services to Use WS-Security</i>	6
4.	<i>Configuring OMII Clients to Use WS-Security</i>	9
5.	<i>Enabling HTTPS in an OMII Server</i>	11
6.	<i>Changing the protocol of an OMII Server</i>	12
7.	<i>Consuming Web Services in HTTPS Servers from an OMII Client</i>	13
8.	<i>HTTPS-Related Advice for Authors of Software Component Installers</i>	15
9.	<i>Implementation of HTTPS in the OMII Container</i>	17
10.	<i>Implementation of HTTPS in the OMII Client</i>	18
11.	<i>Supporting Services that use the GridServIT API</i>	19

1. Introduction

Basic security policies in the OMII Web Services container make use of the *OASIS WS-Security* specifications for Web Services Security, and *HTTPS* — Hypertext Transfer Protocol over the Netscape Secure Sockets Layer (SSL). Individual messages are signed using WS-Security headers — for authentication and non-repudiation. For performance reasons, the typical configuration does not enable *message level encryption* at the WS-Security level — instead it is recommended to enable HTTPS in an OMII container.

This document describes typical WS-Security and HTTPS configurations for Web services in the OMII container, and for clients of those services.

The first part of the document is mainly concerned with WS-Security. The second part mainly deals with HTTPS.

Section 2 contains some background material on X509 Certificates and Keystores in OMII Installations. Section 3 describes how to configure OMII services to use WS-Security. Section 4 does the same for OMII clients.

Section 5 describes how to enable HTTPS in an OMII server installation. Section 6 describes an approach to changing the protocol of an established server. It also includes some information on how the OMII stack installer deals with setting the protocol. Section 7 describes in detail how to use HTTPS from an OMII client. Section 8 contains some advice for those writing installers for software components, where these must support installation into a Web Services Container running HTTPS.

The last 3 sections form appendices. Sections 9 and 10 contain various implementation details for HTTPS handling in server and client installations respectively. Section 11 describes how to configure services that use parts of the GridServIT API.

2. X509 Certificates and Keystores in OMII Installations

Any OMII installation — client or server — will have its own X509 certificate (with associated private key) and a set of certificates for trusted Certificate Authorities. Details of how to access these local certificates and keys are stored in a file called **crypto.properties**. In general you will *not* need to modify this file yourself — it is set up by the OMII installers or the OMII Certificate Management tool. But it is useful to be aware that such a file exists. Both the OMII-customized handlers for WS-Security and the OMII-customized socket factories for HTTPS read this properties file to locate the X509 certificates they require. The **crypto.properties** file should be accessible through the Java class path.

3. Configuring OMII Services to Use WS-Security

The OMII Web services container utilizes a version of Apache Axis. Services are normally configured by a descriptor in Axis Web Services Deployment Descriptor (WSDD) format. A full description of this format is out of scope for this document but see, for example, <http://ws.apache.org/axis/java/reference.html> or <http://www.oio.de/axis-wsdd>.

For purposes of this section, you need to be aware that in Axis a service is normally defined by a particular Java object (here referred to as *the pivot*), which consumes the contents of a request message sent by a client, and generates content for a response message routed back to the client. Within the container, request messages can be directed through a *chain of handlers* before reaching the pivot object. Each handler is itself a Java object, and it can do any kind of pre-processing on the message before it reaches its destination. For example a handler may check that the message is correctly signed. Response messages can also be post-processed by one or more handlers before being sent back to the client.

Handlers before the pivot form the *request flow*; handlers after the pivot form the *response flow*.

WS-Security in OMII is based on Apache WSS4J. What OMII provides is a set of custom Axis handlers, based on handlers in WSS4J, but facilitating common OMII security policies. The most important OMII handler classes are called **WSOutboundHandler** and **PolicyEnforcementHandler**. On the server side, **PolicyEnforcementHandler** sits in the request flow, and **WSOutboundHandler** sits in the response flow (as explained later, this configuration is reversed on the client side).

For the benefit of readers familiar with the standard WSS4J Axis handlers, we note that **WSOutboundHandler** is based on the Apache **WSDoAllSender** and **PolicyEnforcementHandler** is based on the Apache **WSDoAllReceiver**.

For purposes of illustration we assume the pivot is a simple service implemented by a class called **HelloService**. A possible WSDD configuration file for the service is given in Figure 1. The parts we are interested in here are the two `<handler/>` elements — one in the request flow and one in the response flow.

The first `<handler/>` element is implemented by the Java class

uk.ac.omii.security.wss4j.handler.PolicyEnforcementHandler

This handler checks that the incoming message is signed, and that the digital signature covers specified parts of the message. It also stores the X509 certificate that was used for the signature. This certificate identifies the client — it can be retrieved later by the service. In the configuration specified here, **PolicyEnforcementHandler** also checks a timestamp on the incoming message.

The second `<handler/>` element is implemented by the Java class

uk.ac.omii.security.wss4j.WSOutboundHandler

This handler will sign outgoing messages using the keys associated with the X509 certificate of this server. In the configuration here, it will also add a timestamp to the outgoing message.

Unless otherwise stated, the parameters expected by **PolicyEnforcementHandler** are identical to those expected by the Apache WSS4J handler **WSDoAllReceiver**, and parameters expected by **WSOutboundHandler** are identical to those expected by the Apache WSS4J handler **WSDoAllSender**. The parameters appearing in Figure 1 are explained below.

The **action** parameter takes the value “**Timestamp Signature**”. In general the value of the **action** parameter is a space-separated list of security actions for the handlers to perform. The **TimeStamp** action tells **WSOutboundHandler** to add a timestamp to the outgoing message. It tells **PolicyEnforcementHandler** to search for a timestamp to verify that the message was recently created — by default within the last 300 seconds, though this can be configured by setting another parameter called **timeToLive**. The **Signature** action tells **WSOutboundHandler** to sign the outgoing message. It tells **PolicyEnforcementHandler** to check for a signature covering the incoming message. The details of how these signatures are to be handled are controlled by the next few parameters.

```

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
             xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="HelloService" provider="java:RPC">

    <requestFlow>
      <handler type=
        "java:uk.ac.omii.security.wss4j.handler.PolicyEnforcementHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties"/>
        <parameter name="signatureKeyIdentifier" value="DirectReference" />
        <parameter name="passwordCallbackClass"
          value="uk.ac.omii.security.utils.PWCallback"/>
        <parameter name="signatureParts"
          value=
"{{{http://schemas.xmlsoap.org/soap/envelope/}Body;{{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp"
          />
        </handler>
      </requestFlow>

    <responseFlow>
      <handler type=
        "java:uk.ac.omii.security.wss4j.handler.WSOutboundHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties"/>
        <parameter name="signatureKeyIdentifier" value="DirectReference" />
        <parameter name="passwordCallbackClass"
          value="uk.ac.omii.security.utils.PWCallback"/>
        <parameter name="signatureParts"
          value=
"{{{http://schemas.xmlsoap.org/soap/envelope/}Body;{{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp;"
          />
        </handler>
      </responseFlow>

    <parameter name="className" value="HelloService"/>

    <parameter name="allowedMethods" value="*/>

  </service>
</deployment>

```

Figure 1. Possible WSDD configuration for a secure OMII service.

The **signaturePropFile** parameter specifies the name of the file used to locate certificates and keys when signing or checking signatures. As discussed in section 2, this will normally take the value **crypto.properties**.

The **signatureKeyIdentifier** parameter identifies the certificate associated with the signature. For OMII services it is most often set to **DirectReference**, which means that the message itself contains the signing certificate, as a **<BinarySecurityToken/>** element.

The **passwordCallbackClass** parameter specifies a class that is used to find the password protecting private keys associate with the server's X509 certificate. In OMII services it is normally set to **uk.ac.omii.security.utils.PWCallback**. This class actually looks in the **crypto.properties** file to find this password (in future versions of OMII alternative strategies may be supported).

The **signatureParts** parameter specifies which parts of the message to sign (**WSOutboundHandler**), or which parts of the incoming messages must have been signed (**PolicyEnforcementHandler**). It is

a semicolon-separated list of XML *element names*. Note there are two pairs of curly brackets before each local element name. The first pair of curly brackets is actually redundant in **signatureParts** and left empty (it is used in **encryptionParts**, not illustrated here). The second pair of curly brackets contains the XML namespace for the element. In the example the elements of the message that are signed are the **<Body/>** element of the SOAP message, namespace

http://schemas.xmlsoap.org/soap/envelope/

and the **<TimeStamp/>** element, namespace

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd

If the **signatureParts** parameter is omitted, the default policy is that just the **<Body/>** element should be signed.

Signing a request message is the preferred way for an OMII client to authenticate itself to an OMII service. The “business logic” of the service will often need to know the client’s identity. OMII provide a simple API that allows a service implementation (or for that matter any other handler downstream of **PolicyEnforcementHandler**) to access this information. The example implementation of **HelloService** in Figure 2 illustrates the use of this API.

The static method **getCurrentUser()** in the class **AuthenticatedCertificateHelper** returns the X509 certificate associated with the signature verified by **PolicyEnforcementHandler**.

The example also illustrates use of the convenience class **DNParser**, which can be used for extracting fields from the X509 *Distinguished Name*.

This API, and the handler classes discussed earlier in this section, live in a JAR file called

omii-security-utils-X.Y.jar

This archive must be accessible through the class loader of the service. In a normal OMII server installation, this file is present in the **shared/lib/** directory of the Tomcat installation. The **crypto.properties** file must also be accessible through the class loader, and in an OMII installation is normally present in the **shared/classes/** directory of the Tomcat installation.

```
import uk.ac.omii.security.wss4j.helpers.AuthenticatedCertificateHelper ;
import uk.ac.omii.security.utils.DNParser;

import java.security.cert.X509Certificate;

public class HelloService {

    public String hello() throws Exception {

        X509Certificate clientCert =
            AuthenticatedCertificateHelper.getCurrentUser() ;

        DNParser dn = new DNParser(clientCert.getSubjectDN().getName());

        return "Hello " + dn.getCN() + "!" ;

    }
}
```

Figure 2. Possible implementation of HelloService.

4. Configuring OMII Clients to Use WS-Security

Axis clients can be configured to use WS-Security using the WSDD file format, in a way very similar to Axis services. A typical client configuration is reproduced in Figure 3.

As advertised, this configuration has much in common with the service configuration in Figure 1. Again it uses the handlers **WSOutboundHandler** and **PolicyEnforcementHandler**. But now the former is in the request flow and the latter is in the response flow. Notice that the request and response flow are now defined in a **<globalConfiguration/>** element. This is common practise when defining a client configuration (though not mandatory).

This client configuration causes request messages sent by the client to be timestamped and signed. It also ensures that service-generated signatures and timestamps on incoming response messages are validated. Most of the details are very similar to the server side example described in section 3, and we refer the reader back to that section.

One new feature in this example is the introduction of the parameter **ignoreEndpointCNmismatch**. Without this parameter, **PolicyEnforcementHandler** will throw an exception if the *Common Name* associated with the server certificate in the response is *not identical* to the *DNS name* of that server. In many cases it is good policy to require the common name to be the same as the DNS name, but in some cases it can be restrictive, so the option to disable this check is useful.

The appearance of the transport **OMIIHTTPSender** will be discussed in section 7.

There are various ways to put a client configuration like the one in Figure 3 into effect. One way is to simply copy this WSDD to a file called **client-config.wsdd**, and place this file somewhere on the class path of the client application. The standard OMII client installation contains just such a file in the directory **OMIIClient/conf/**, so you may simply add this directory to your class path.

Experience suggests that this approach often leads to problems — for example one may pick up the wrong client configuration when there are several **client-config.wsdd** files in different jars on the class path. A better approach is to define the Java system property **axis.ClientConfigFile**, setting its value to the path to a WSDD file. For example you may start your client by something like:

```
$ java -Daxis.ClientConfigFile=deploy-myclient.wsdd MyClient
```

An even safer approach is to define the WSDD configuration on an invocation-by-invocation basis *within* the client code. For example your client code may contain:

```
import org.apache.axis.EngineConfiguration ;
import org.apache.axis.configuration.FileProvider ;

import javax.xml.rpc.Service ;
...

EngineConfiguration config = new FileProvider("deploy-myclient.wsdd") ;
Service service = new org.apache.axis.client.Service(config) ;
```

...then use this **Service** object to make JAX RPC calls in the usual way.

In any case, to use the handlers described in this section, the JAR file

```
omii-security-utils-X.Y.jar
```

must be available on your class path. In a normal OMII client installation, this file is present in the **OMIIClient/lib/** directory. The **crypto.properties** file must also be accessible through the class path, and in an OMII client installation it is normally present in the **OMIIClient/conf/** directory.

```

<?xml version='1.0'?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <globalConfiguration>

    <requestFlow>
      <handler type=
        "java:uk.ac.omii.security.wss4j.handler.WSOutboundHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties"/>
        <parameter name="signatureKeyIdentifier" value="DirectReference"/>
        <parameter name="signatureParts"
          value=
"{{{http://schemas.xmlsoap.org/soap/envelope/}Body;{{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp"
          />
        <parameter name="passwordCallbackClass"
          value="uk.ac.omii.security.utils.PWCallback"/>
      </handler>
    </requestFlow>

    <responseFlow>
      <handler type=
        "java:uk.ac.omii.security.wss4j.handler.PolicyEnforcementHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties" />
        <parameter name="signatureKeyIdentifier" value="DirectReference" />
        <parameter name="passwordCallbackClass"
          value="uk.ac.omii.security.utils.PWCallback"/>
        <parameter name="signatureParts"
          value=
"{{{http://schemas.xmlsoap.org/soap/envelope/}Body;{{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp;"
          />
        <parameter name="ignoreEndpointCNmismatch" value="true"/>
      </handler>
    </responseFlow>

  </globalConfiguration>

  <transport name="http" pivot="java:uk.ac.omii.transport.http.OMIIHTTPSender"/>
</deployment>

```

Figure 3. Typical WSDD configuration for a secure OMII client.

5. Enabling HTTPS in an OMII Server

The preferred way to enable HTTPS in an OMII server is to specify this at installation time. Switching an established OMII server between HTTP and HTTPS is of course possible (see Section 6), but not highly recommended.

Before running the OMII server stack installer to install the Web Services container, define an environment variable **OMII_TOMCAT_PROTOCOL**. The supported values for this variable are **http** or **https**, both in lower case. In OMII 3.1.0 the default value is **http**.

If you are going to be using HTTPS, you will probably also want to change the port number on which the server listens. Its default value is “**18080**”. It is not essential to change this, but by popular convention HTTPS servers often listen on port numbers ending in the digits “**443**”. You can change the port by setting the **OMII_TOMCAT_PORT** environment variable before running the stack installer.

So a possible installation of the Web Services Container begins:

```
$ export OMII_TOMCAT_PROTOCOL=https
$ export OMII_TOMCAT_PORT=18443
$ ./OMIIstackInstall.pl
[...]  
> 2  
[...]
```

The value “2” entered here is selecting installation of the Web Services Container components.

On successful completion of this phase of installation, you should be able to see the home page of your installation by visiting the URL:

https://<server-name>:18443

with a Web browser. If you receive a warning because the server’s certificate was not issued by an authority trusted by your browser, take the option to proceed, accepting the server’s certificate.

6. Changing the protocol of an OMII Server

As described in section 5, the recommended procedure for setting up an HTTPS-enabled OMII Server is to set the **OMII_TOMCAT_PROTOCOL** environment variable before invoking the OMII stack installer to install the Web Services Container.

In OMII 3.1.0, the way this is actually implemented is that the stack installer first installs the container in the “traditional” way using the HTTP protocol. Then it switches the protocol to HTTPS (if that is what is selected by the **OMII_TOMCAT_PROTOCOL** variable) using a script called **OMIIswitchProtocol.pl**.

The **OMIIswitchProtocol.pl** script can be found in the **extension/** subdirectory of the OMII server distribution. It must be invoked from the top level directory of the distribution. Before invoking this script, four environment variables:

```
OMII_TOMCAT_PROTOCOL  
OMII_TOMCAT_PORT  
OMII_HOME  
OMII_ANT
```

must be set. The first two have been described earlier; the value of **OMII_HOME** should be the path to the target directory where the OMII server has been installed; the value of **OMII_ANT** should be the path to the executable **ant** command for an OMII-supplied version of Apache Ant.

If you want to use the script **OMIIswitchProtocol.pl** directly to switch the protocol of a Web Services Container installation, go to the top-level directory of an OMII server distribution, and run something like:

```
$ export OMII_TOMCAT_PROTOCOL=https  
$ export OMII_TOMCAT_PORT=18443  
$ export OMII_HOME=/home/omii/OMII  
$ export OMII_ANT=/home/omii/omii-server-3.1.0/apache-ant-1.6.1/bin/ant  
  
$ extension/OMIIswitchProtocol.pl
```

You can use this procedure to switch from HTTP to HTTPS or vice versa. But be aware that this script only modifies the basic Web Service Container. If additional software components have been installed within the container, it is possible that they may have dependencies on the protocol that was in effect *at the time they were installed*. Whether or not this is the case depends on how the component was written.

The **OMIIswitchProtocol.pl** script does two things:

- It runs an XSLT script that replaces the existing **<Connector/>** element in the Tomcat “**server.xml**” file with one appropriate to the selected protocol (see Figure 6 for an example of this).
- It edits the file **<OMII_HOME>/records/omii**, updating the values of two properties **tomcat.protocol** and **tomcat.port**. Subsequent invocations of the stack installer — for example to install additional software components — look in this file for properties of the OMII installation.

If preferred, these files can be edited manually, rather than by using the script. But of course this needs some care.

The OMII server will normally have to be restarted for these changes to take effect.

7. Consuming Web Services in HTTPS Servers from an OMII Client

In most respects accessing Web Services on HTTPS servers is identical to accessing services on HTTP servers. The most obvious difference is that the URL scheme changes from “**http:**” to “**https:**”. But there are also technical issues connected with access to trusted X.509 certificates and private keys.

When connecting to an HTTPS server, a Java client will normally insist that the X.509 certificate presented by the server is “trusted”. To establish this, it needs to access a store of trusted root certificates — a *trust store*.

If the server demands HTTPS-based *client-side authentication*, the client will also need access to the private key associated with its own X.509 certificate. To retrieve this, a Java client needs to access a store of locally held private keys — a *key store*. OMII servers do *not* demand *HTTPS-based* client-side authentication by default, but they can be configured to do so — see section 9.

In OMII 3.3.0, any client that consumes Web Services on HTTPS servers will automatically pick up certificates and keys needed for HTTPS authentication, *provided* it uses an Axis client configuration with **OMIIHTTPSender** as the transport handler. This was already illustrated in Figure 3. Because this configuration contains the line:

```
<transport name="http" pivot="java:uk.ac.omii.transport.http.OMIIHTTPSender"/>
```

it will automatically pick up the required credentials when communicating with an HTTPS server.

Note that if you want to use HTTPS *but not* WS-Security, you must still specify **OMIIHTTPSender** as the transport handler. Figure 4 contains a possible client configuration *without* the WS-Security handlers, but supporting the OMII 3.3.0 style of HTTPS handling.

When using the *standard* transport handlers provided by Apache, credential stores are usually specified by setting appropriate values for the Java system properties:

```
javax.net.ssl.keyStore  
javax.net.ssl.keyStorePassword  
javax.net.ssl.trustStore  
javax.net.ssl.trustStorePassword
```

As explained above, the **OMIIHTTPSender** has been written so that it automatically uses the central OMII key store to load these credentials. If you are using this transport handler, it is *unnecessary* to set the values of the above-mentioned properties; moreover if you *do* set them, their values will be *ignored*!

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultClientConfig"
  xmlns=http://xml.apache.org/axis/wsdd/
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <globalConfiguration>
    <parameter name="disablePrettyXML" value="true"/>
  </globalConfiguration>

  <transport name="http"
    pivot="java:uk.ac.omii.transport.http.OMIIHTTPSender"/>

  <transport name="local"
    pivot="java:org.apache.axis.transport.local.LocalSender"/>

  <transport name="java"
    pivot="java:org.apache.axis.transport.java.JavaSender"/>

</deployment>
```

Figure 4. Possible WSDD configuration for OMII client without WS-Security, but supporting HTTPS.

8. HTTPS-Related Advice for Authors of Software Component Installers

Starting with OMII 3.1.0, authors writing installer code for OMII software components should endeavour to support installation into an OMII Web Services Container that may be running the secure HTTPS protocol.

As described in the document *MP Integration Specification for OMII 2.x & OMII 3.x*, authors of a software component for the OMII container should provide an Apache Ant build script, exporting a set of targets that will be invoked during the life cycle of the component — for example during installation of the component into an OMII container.

When targets from this build file are invoked from the master OMII installer, a set of predefined properties are passed in to Ant. These include properties like **omii.server.home**, **tomcat.port**, **tomcat.host**, etc. The complete list can be found in the above-mentioned integration specification.

Starting with OMII 3.1.0, the set of properties passed to software component build and installation targets is supplemented with **tomcat.protocol**. This takes one of the two values “**http**” or “**https**” (in lower case). Where an Ant target needs to connect to the Web Services Container, it should take this protocol into account.

For example, prior to OMII 3.1.0, an Ant target in the software component build file might access the Web Services Container through a URL constructed as follows:

```
http://${tomcat.host}:${tomcat.port}/path
```

As of OMII 3.1.0, the corresponding URL could be constructed as follows:

```
${tomcat.protocol}://${tomcat.host}:${tomcat.port}/path
```

In upgrading software component installation scripts to support HTTPS, a useful rule of thumb is to search for the string “**tomcat.port**”. Experience suggests that most often this string appears in a URL, and most often that URL should be changed to use the protocol **\${tomcat.protocol}**!

Native Ant tasks that access URLs — `<get/>`, the `<http/>` test of `<condition/>`, etc — should be able to handle these HTTPS URLs without problems, because the master OMII installer now initializes the Java system property **javax.net.ssl.trustStore** at the time the Ant process is started (in fact by setting the **ANT_OPTS** environment variable). This gives these tasks access to a credential store for authenticating the server.

Where an Ant target accesses the Web Services Container indirectly, e.g. by forking another process (Java or non-Java) this is unlikely to be sufficient. Other means must be used to provide access to relevant credential stores.

Suppose, for example, the Ant target uses the `<java/>` task, setting the **fork** attribute to “**yes**”. In particular consider the case where the target needs to use the Axis **AdminClient** class to deploy a service to a running container. A suitable `<java/>` task is illustrated in Figure 5. The important point here is that we explicitly set the **javax.net.ssl.truststore** system property in the forked Java process. Its value references the central key store for this OMII installation.

(In this particular example, **AdminClient** is itself an Axis client. So an alternative would be to specify an Axis client configuration like the one in Figure 4 for this task.)

Where the server has been configured to require HTTPS-based *client authentication*, the situation can be more complicated. For the present it may be acceptable to insist that this feature is disabled while a new software component is being installed.

```
<java classname="org.apache.axis.client.AdminClient"
      fork="yes" failonerror="true">
  <classpath ../>
  <arg value =
    "-l${tomcat.protocol}://localhost:${tomcat.port}/axis/services/AdminService"/>
  <arg value="server-config.wsdd"/>
  <sysproperty key="javax.net.ssl.trustStore"
    value="${omii.server.home}/omii.ks"/>
</java>
```

Figure 5. <java/> Task Providing Access to OMII Credential Store

9. Implementation of HTTPS in the OMII Container

The `conf/server.xml` configuration file for an OMII Tomcat container that has been configured to use the HTTPS protocol should include a `<Connector/>` element similar to the one illustrated in Figure 6. The most interesting feature here is the specification of a non-default implementation in the `SSLImplementation` property.

```
<!-- Define an OMII-style SSL Coyote HTTP/1.1 Connector on port 18443 -->
<Connector port="18443"
    maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
    enableLookups="false" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https" secure="true"
    clientAuth="false" sslProtocol="SSL"
    SSLImplementation="uk.ac.omii.ssl.OMIIImplementation"/>
```

Figure 6. Connector element in “server.xml”, specifying OMII-style HTTPS

The class `uk.ac.omii.ssl.OMIIImplementation` lives in the jar file:

`omii-ssl-1.0.jar`

The source of this class is in `source/omii-deployment/omii-ssl/` in the OMII distributions. It is an implementation of the interface `org.apache.tomcat.util.net.SSLImplementation`.

The most important method in `OMIIImplementation` class is the method `getServerSocketFactory()`, which returns an `org.apache.tomcat.util.net.ServerSocketFactory` instance. The socket factory returned (through an intermediary `org.apache.tomcat.util.net.jsse.OMIIFactory`) is actually an instance of `org.apache.tomcat.util.net.jsse.OMIISocketFactory`.

The default Tomcat `SSLImplementation` would have returned an instance of `org.apache.tomcat.util.net.jsse.JSSE14SocketFactory`. The OMII variant differs in the way it handles trust stores and key stores. It loads the description of these credential stores from a file called `crypto.properties`. Currently this file is assumed to live in a directory located at `“./grid”`, relative to `CATALINA_HOME`.

As well as reading key store parameters from an OMII `crypto.properties` file, this factory differs from the standard Tomcat factory in allowing the *key store password* to be distinct from the *passwords of any private keys* within that key store. This is considered to be a security enhancement.

[The current implementation of `OMIISocketFactory` reuses much code from Tomcat’s own `JSSE14SocketFactory`. Reasonably enough it extends this class. The current implementation actually reuses parts of the code in the super class that have *package-level access*. For this reason it is placed in the `org.apache.tomcat.util.net.jsse` package, although it is not part of the standard Tomcat distribution, and it physically lives in a separate, OMII-sourced, jar file. But this means this class is more tightly coupled to details of the Tomcat implementation than one would ideally prefer (which is bad). On the other hand, it maximizes code reuse (which is good)...]

OMII installers set the `ClientAuth` property to `false`. This means that the server does *not* demand *HTTPS-based* authentication of the client (by X.509 certificate). Recall that OMII clients are usually authenticated at the WS-Security level, by message signing. Change the `clientAuth` attribute on the `<Connector/>` element to `true` if you want your clients to be authenticated at the HTTPS transport level; but be aware not all clients of an OMII server will necessarily be *able* to authenticate themselves this way.

10. Implementation of HTTPS in the OMII Client

An (Axis-based) OMII client that uses HTTPS to communicate with a Web Services container should have a client configuration WSDD file that specifies **uk.ac.omii.transport.http.OMIIHTTPSender** as the transport handler, as illustrated in Figure 3 and Figure 4. This class locates keys and certificates for mutual HTTPS authentication from a client's central OMII key store.

The normal Axis **CommonsHTTPSender** uses code from the *Apache Commons HTTP Client* to communicate with a server. The OMII version pre-registers a **org.apache.commons.httpclient.protocol.Protocol** for the **https** protocol. This **Protocol** object includes a custom socket factory of class

uk.ac.omii.security.ssl.OMIIProtocolSocketFactory

This class is an implementation of the interface:

org.apache.commons.httpclient.protocol.SecureProtocolSocketFactory

Internally it uses a standard **javax.net.ssl.SSLSocketFactory** to create sockets. This factory is created through a **javax.net.ssl.SSLContext**. This **SSLContext** is initialized using custom **KeyManager** and **TrustManager** returned by the class **uk.ac.omii.security.ssl.OMIISSLCrypto**.

The **OMIISSLCrypto** class exposes two static methods: **getKeyManagers()** and **getTrustManagers()**. These return arrays of **javax.net.ssl.KeyManager** and **javax.net.ssl.TrustManager** respectively — each array normally containing a single element. During initialization, **OMIISSLCrypto** loads an OMII-style **crypto.properties** file from the class path. The key manager and trust managers returned by this class are constructed on the basis of the properties read from this file. The returned key manager (in particular) is of a custom class — **OMIISSLCrypto.OMIIKeyManager** — that allows the *key store password* to be distinct from the *passwords of private keys* within that key store. This is considered to be a security enhancement. (Parenthetically, it also allows the key store to hold more than one private key, though only one is selected).

The source for the classes described here can be found in **source/omii-security-utils/** in the OMII distributions.

[The customization to the Tomcat socket factory described in Section 9 is older code than the customization to the client socket factory described here. In future releases from OMII, the OMII-customized Tomcat classes are likely to be re-factored to exploit code described in this section — e.g. **OMIISSLCrypto**.]

11. Supporting Services that use the GridServIT API

OMII services may extend the class

uk.ac.soton.itinnovation.grid.gridservit.EScienceService

and use API associated with this class to access contextual information, such as the X509 certificate of the currently active client. To do this in the current version of OMII it is necessary to add two extra handlers to the request flow. An example WSDD configuration is given in Figure 7,

```

<deployment xmlns="http://xml.apache.org/axis/wsdd/"
             xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="HelloService" provider="java:RPC">

    <requestFlow>
      <handlertype =
"java:uk.ac.soton.itinnovation.grid.gridservit.axis.handlers.ServiceContextInitHand
ler"
      />
      <handler type=
"java:uk.ac.omii.security.wss4j.handler.PolicyEnforcementHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties"/>
        <parameter name="signatureKeyIdentifier" value="DirectReference" />
        <parameter name="passwordCallbackClass"
value="uk.ac.omii.security.utils.PWCallback"/>
        <parameter name="signatureParts"
value=
"{{http://schemas.xmlsoap.org/soap/envelope/}}Body;{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp"
        />
      </handler>
      <handler type=
"java:uk.ac.soton.itinnovation.grid.gridservit.axis.handlers.IntialiseServiceContex
tHandler"
      />
    </requestFlow>

    <responseFlow>
      <handler type=
"java:uk.ac.omii.security.wss4j.handler.WSOutboundHandler">
        <parameter name="action" value="Timestamp Signature"/>
        <parameter name="signaturePropFile" value="crypto.properties"/>
        <parameter name="signatureKeyIdentifier" value="DirectReference" />
        <parameter name="passwordCallbackClass"
value="uk.ac.omii.security.utils.PWCallback"/>
        <parameter name="signatureParts"
value=
"{{http://schemas.xmlsoap.org/soap/envelope/}}Body;{{http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd}Timestamp;"
        />
      </handler>
    </responseFlow>

    <parameter name="className" value="HelloService"/>

    <parameter name="allowedMethods" value="*" />

  </service>
</deployment>

```

Figure 7. Possible WSD configuration for a GridServIT service.