AGIV Java Security Developer's Guide

Integrating the AGIV web services in your Java applications.
Version 1.0.1

Frank Cornelis

25 May 2013

Abstract

The AGIV web services use Windows Communication Foundation to setup the security constraints and requirements. Connecting to WCF secured web services from within Java can be a challenge because the support for WS-* within JAX-WS web service stacks has not yet been standardized through the JCP. Most of the time it is also not so trivial to switch to JAX-WS web service stack implementations that do support WS-* (and that work) for connecting clients. Via the AGIV Java Security framework we provide an alternative solution to turn a vanilla JAX-WS client into an AGIV Security enabled client.

1. Introduction

The AGIV security architecture is based on a quite modern WS-Trust based system. This system allows for a very flexible IAM solution. The architecture includes the following services:

- A WS-Trust based IP-STS service (IP = Identity Provider) that issues security tokens based on username/password or certificate credentials.
- A WS-Trust based R-STS service (R = Relaying/Resource) that issues security tokens based on the IP-STS security tokens. The R-STS security tokens always apply to a certain AGIV web service.
- Each AGIV web service requires either the setup of a WS-SecureConversation based secure conversation or directly use the SAML token as retrieved from the R-STS.

The AGIV web services are secured via WS-SecureConversation or directly via a SAML token. The security policy is defined via WS-Policy and WS-SecurityPolicy and can be found within the WSDLs of the different web services. For using such secured web services from within a Java application different options are available:

- One can simply use the WS-* features of your WS-* enabled web service stack. This requires a proprietary configuration that is specific to the used web service stack. It is important to understand that this developer's guide does not describe how to configure each and every WS-* enabled web service stack.
- One can disable the WS-* features of your WS-* enabled web service stack and use the AGIV Security framework. Not all web service stacks allow you to disable the WS-* functionality. Most will automatically kick in the WS-* features once a WS-Policy enabled binding is found in the WSDL.
- Use a vanilla web service stack that has no notion of WS-* and use the AGIV Security framework.
- Remove the WS-Policy entries from the WSDL and use the AGIV Security framework. This is probably the easiest solution and works for all JAX-WS web service stacks.

2. JAX-WS runtime

The AGIV Security framework does not use WS-* functionality of any JAX-WS runtime. The AGIV Security framework also does not require a specific JAX-WS runtime to be available. The AGIV Security framework strictly uses the plain vanilla JAX-WS API to be able to run on as many JAX-WS runtimes as possible. The *Table 1, “Tested JAX-WS runtimes”* summarizes the JAX-WS runtimes and their versions on which the AGIV Security framework has been tested.

Table 1. Tested JAX-WS runtimes

JAX-WS runtime	Java runtime		
	Java 1.5.0_22	Java 1.6.0_45	Java 1.7.0_21
Default JAX-WS	[a]		
JAX-WS RI	2.1.7, 2.1.9, 2.2.5 [b]	2.1.7, 2.1.9, 2.2.6, 2.2.6-2, 2.2.7	2.1.7, 2.1.9, 2.2.6, 2.2.6-2, 2.2.7
Apache CXF [c]	2.3.9, 2.3.10, 2.3.11, 2.4.6, 2.4.7, 2.4.8, 2.4.9, 2.4.10, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.6.6, 2.6.7, 2.6.8 [d]	2.3.9, 2.3.10, 2.3.11, 2.4.6, 2.4.7, 2.4.8, 2.4.9, 2.4.10, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.6.6, 2.6.7, 2.6.8, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5	2.3.9, 2.3.10, 2.3.11, 2.4.6, 2.4.7, 2.4.8, 2.4.9, 2.4.10, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.6.5, 2.6.6, 2.6.7, 2.6.8, 2.7.0, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5
Axis2	1.6.1, 1.6.2	[e]	1.6.1, 1.6.2
Metro [f]	2.1.1 [g]	2.1.1, 2.2, 2.2.0-1, 2.2.0-2, 2.2.1, 2.2.1-1	2.1.1, 2.2, 2.2.0-1, 2.2.0-2, 2.2.1, 2.2.1-1

[a] Not applicable.

[b] JAX-WS RI versions 2.2.6, 2.2.6-2, 2.2.6-3, 2.2.7 no longer run on Java 1.5.

[c] Need to use WSDLs without WS-Policy or disable WS-Policy feature. Disabling WS-Policy feature does not work for version 2.6.0+. Version 2.6.0+ also requires removal of the `<wsp:Policy>` elements, else the WS-Policy parser kicks in. https://issues.apache.org/jira/browse/CXF-4258

[d] Apache CXF 2.7 no longer runs on Java 1.5.

[e] Conflicts with the default JAX-WS 2.1 API if you use the JAX-WS WS-Addressing feature.

[f] Need to use WSDLs without WS-Policy.

[g] Metro versions 2.2, 2.2.0-1, 2.2.0-2, 2.2.1, 2.2.1-1 do not run on Java 1.5.

3. Usage

First of all you need to generate JAX-WS stubs out of the WSDL of the service that you want to use. To make sure that your JAX-WS web service stack does not start to interpret the WS-Policy section of the WSDL, you can simply remove the `<wsp:PolicyReference>` element from the `<wsdl:Binding>` element(s) within the WSDL. As some JAX-WS stacks even start to interpret the WS-Policy section after removal of the `<wsp:PolicyReference>` elements, it might be required to even remove the `<wsp:Policy>` elements altogether.

Basically you decorate a vanilla JAX-WS stub to enable the AGIV security on it.

```
import javax.xml.ws.soap.AddressingFeature;
import be.agiv.security.AGIVSecurity;
```

```
import javax.xml.ws.BindingProvider;

// create your JAX-WS stub (generated out of a WSDL)
YourService yourJaxWsClientStub = ...;
YourServicePort yourJaxWsClientPort = yourJaxWsClientStub.getYourPort(
    new AddressingFeature());

// create an AGIV security component
AGIVSecurity agivSecurity = new AGIVSecurity(
    "https://the.location.of.the.agiv.ipsts.service",
    "https://the.location.of.the.agiv.rsts.service",
    "realm",
    "yourUsername", "yourPassword");

// enabled the AGIV security framework on your JAX-WS stub
BindingProvider bindingProvider = (BindingProvider) yourJaxWsClientPort;
agivSecurity.enable(bindingProvider,
    "https://the.location.of.the.agiv.service",
    "serviceRealm");

// ready for usage
yourJaxWsClientPort.invokeBusinessMethod(...);
```

Make sure to enable the JAX-WS WS-Addressing feature and to use the SOAP 1.2 binding. The SOAP 1.2 binding can be enabled by compiling the WSDL via `wsimport` with the `extension` attribute set to `true` .

3.1. AGIV STS Locations and Realms

The location of the AGIV Beta IP-STS service for username/password credentials is:

`https://auth.beta.agiv.be/ipsts/Services/DaliSecurityTokenServiceConfiguration.svc/IWSTrust13`

The location of the AGIV Beta IP-STS service for certificate credentials is:

`https://auth.beta.agiv.be/ipsts/Services/DaliSecurityTokenServiceConfiguration.svc/CertificateMessage`

The location of the AGIV Beta R-STS service is:

`https://auth.beta.agiv.be/sts/Services/SalvadorSecurityTokenServiceConfiguration.svc/IWSTrust13`

The AGIV Beta R-STS realm is

`urn:agiv.be/salvador`

You can use the `AGIVSecurity.BETA_REALM` constant for this.

The AGIV production R-STS realm is

`urn:agiv.be/sts`

You can use the `AGIVSecurity.PRODUCTION_REALM` constant for this.

3.2. Service Locations and Realms

The *Table 2, "AGIV Service Locations and Realms"* lists the locations and realms for the different AGIV services.

Table 2. AGIV Service Locations and Realms

Gipod Service	
Beta Endpoint	`https://gipod.beta.agiv.be/WebService/GipodService.svc`
Beta Realm	`urn:agiv.be/gipodbeta`
Production Endpoint	`https://gipod.agiv.be/WebService/GipodService.svc`
Production Realm	`urn:agiv.be/gipod`
RVV service	
Beta Endpoint	`https://rvv.beta.agiv.be/WebService/RVVService.svc`
Beta Realm	`urn:agiv.be/rvv`
Production Endpoint	`https://rvv.agiv.be/WebService/RVVService.svc`
Production Realm	`urn:agiv.be/rvv`
RVV Upload service	
Beta Endpoint	`https://rvv.beta.agiv.be/WebService/uploadService.svc`
Beta Realm	`urn:agiv.be/rvv`
Production Endpoint	`https://rvv.agiv.be/WebService/uploadService.svc`
Production Realm	`urn:agiv.be/rvv`
Crab WST	
Beta Endpoint	`https://crab.beta.agiv.be/WST/CRAB_WST.svc`
Beta Realm	`urn:agiv.be/crab/beta`
Production Endpoint	`https://crab.agiv.be/WST/CRAB_WST.svc`
Production Realm	`urn:agiv.be/crab`
Crab Read service	
Beta Endpoint	`https://crab.beta.agiv.be/read/crabreadservice.svc`
Beta Realm	`urn:agiv.be/crab/beta`
Production Endpoint	`https://crab.agiv.be/read/crabreadservice.svc`
Production Realm	`urn:agiv.be/crab`
Crab Edit service	
Beta Endpoint	`https://crab.beta.agiv.be/edit/crabeditservice.svc`

Crab Edit service	
Beta Realm	`urn:agiv.be/crab/beta`
Production Endpoint	`https://crab.agiv.be/edit/crabeditservice.svc`
Production Realm	`urn:agiv.be/crab`
Crab Melding service	
Beta Endpoint	`https://crab.beta.agiv.be/melding/crabmeldingservice.svc`
Beta Realm	`urn:agiv.be/crab/beta`
Production Endpoint	`https://crab.agiv.be/melding/crabmeldingservice.svc`
Production Realm	`urn:agiv.be/crab`
Crab OGC service	
Beta Endpoint	`https://crab.beta.agiv.be/ogc/service.svc`
Beta Realm	`urn:agiv.be/crab/beta`
Production Endpoint	`https://crab.agiv.be/ogc/service.svc`
Production Realm	`urn:agiv.be/crab`

3.3. Dependencies

The `AGIVSecurity` component can be found within the `agiv-security-client` JAR artifact. Also make sure to include all required dependencies, which are located under the `lib/` directory within the SDK package.

In case you use Maven as build system, you can add the following dependency to your project:

```
<dependency>
    <groupId>be.agiv.security</groupId>
    <artifactId>agiv-security-client</artifactId>
    <version>1.0.1</version>
</dependency>
```

The AGIV Java Security framework artifacts are available in the public e-contract.be Maven repository. Add the folowing repository configuration to the `repositories` element of your `pom.xml` Maven POM file:

```
<repository>
    <id>e-contract.be</id>
    <url>http://www.e-contract.be/maven2/</url>
    <releases>
        <enabled>true</enabled>
    </releases>
</repository>
```

3.4. WS-SecureConversation

Per default the `AGIVSecurity` component will not use WS-SecureConversation on the JAX-WS stubs. For AGIV web services that support WS-SecureConversation you can enable the usage of WS-SecureConversation on the JAX-WS stubs by using the different variants of the `enable` method.

```
BindingProvider bindingProvider = (BindingProvider) yourJaxWsClientPort;
boolean useWsSecureConversation = true;
agivSecurity.enable(bindingProvider,
    "https://the.location.of.the.agiv.service",
    useWsSecureConversation, "serviceRealm");
```

3.5. Certificate credential

The `AGIVSecurity` component also supports X509 certificates as user credential. The X509 certificate and corresponding private key are most likely stored within a PKCS#12 keystore. Under *Section 4, "Managing Certificate Credentials"* we describe how to load certificate credentials from software key stores or from your eID card. The `AGIVSecurity` component can load the key material from the keystore file as follows:

```
import java.io.File;
import be.agiv.security.AGIVSecurity;

File pkcs12File = new File("/the/location/of/the/pkcs12/keystore");
AGIVSecurity agivSecurity = new AGIVSecurity(
    "https://the.location.of.the.agiv.ipsts.service",
    "https://the.location.of.the.agiv.rsts.service",
    "realm",
    pkcs12File, "pkcs12Password");
```

In case that the X509 certificate and corresponding private key is stored on some other medium (like for example an HSM or a smart card) you can use the following constructor.

```
import java.security.cert.X509Certificate;
import java.security.PrivateKey;
import be.agiv.security.AGIVSecurity;

X509Certificate certificate = ...;
PrivateKey privateKey = ...;
AGIVSecurity agivSecurity = new AGIVSecurity(
    "https://the.location.of.the.agiv.ipsts.service",
    "https://the.location.of.the.agiv.rsts.service",
    "realm",
```

```
    certificate, privateKey);
```

Java comes with different security providers for loading key material like for example the `SunPKCS11` security provider for loading key material via native PKCS#11 modules.

3.6. Token caching

The `AGIVSecurity` component will cache the different security tokens from the IP-STS, R-STS, and (if applicable) the secure conversation tokens. When the security tokens are about to expire, the `AGIVSecurity` component will automatically retrieve new security tokens.

Although JAX-WS stubs themselves are not multi-threaded, the `AGIVSecurity` component can be shared between multiple threads. Doing so all JAX-WS stubs will use the same security tokens when accessing the AGIV web services.

To improve user experience you can let the `AGIVSecurity` component prefetch the required security tokens for a certain AGIV web service via:

```
agivSecurity.prefetchTokens("https://the.location.of.the.agiv.service",
    "serviceRealm");
```

To reduce the load on the AGIV web services you can cancel the secure conversation tokens after usage via:

```
agivSecurity.cancelSecureConversationTokens();
```

3.7. Web Proxies

The `AGIVSecurity` component supports both HTTP and SOCKS based web proxies. The proxy settings can be configured as follows:

```
import java.net.Proxy.Type;

agivSecurity.setProxy("localhost", 3128, Type.SOCKS);
```

The proxy settings apply to both the different WS-Trust clients that are internally used, as well as to the AGIV Security enabled JAX-WS stubs.

3.8. External IP-STS

The `AGIVSecurity` component also offers support for external IP-STS services that do not behave exactly like the AGIV IP-STS service. The `AGIVSecurity` component can be configured to use an external IP-STS client as follows:

```
import be.agiv.security.AGIVSecurity;

AGIVSecurity agivSecurity = new AGIVSecurity(
    externalIpStsClient, "https://the.location.of.the.agiv.rsts.service");
```

Where `externalIpStsClient` is your own custom IP-STS client object that implements the `ExternalIPSTSClient` interface. The external IP-STS client should be thread-safe.

3.9. Activity listener

When the `AGIVSecurity` cache is empty, or the cached tokens are about to expire, it can take a while to acquire new tokens due to the number of STS calls and cryptographic operations that need to be performed. Especially when using the `AGIVSecurity` component within a desktop Java (Swing) application this could impact the end user experience dramatically.

To be able to give the end user an indication of what is taking so long, the application can register an `STSListener` on the `AGIVSecurity` component. Via the registered `STSListener` the application will receive a callback whenever the `AGIVSecurity` component in requesting a new token.

3.10. Javadoc

The javadoc API documentation can be found within the `agiv-security-client-javadoc` directory within the SDK package.

4. Managing Certificate Credentials

In this chapter we describe how to manage either software certificate credentials or eID based certificate credentials.

4.1. Software Certificates

4.1.1. Creating a key store

In this section we describe how to create a PKCS#12 key store containing a self-signed certificate using OpenSSL (on Linux).

Create a 1024 bit RSA key pair with default public exponent via:

```
openssl genrsa -out key.pem -F4 1024
```

From this RSA key pair we next create a new self-signed certificate via:

```
openssl req -config openssl.conf -new -x509 -key key.pem -out cert.pem -verbose
 -days 365
```

with the configuration file `openssl.conf` containing:

```
[req]
distinguished_name = req_distinguished_name
prompt = no
x509_extensions = req_x509_extensions

[req_distinguished_name]
commonName=JavaTestAccount

[req_x509_extensions]
```

You can view the content of this new certificate via:

```
openssl x509 -noout -text -in cert.pem
```

Finally you can create a PKCS#12 key store containing both the certificate and the corresponding private key via:

```
openssl pkcs12 -export -out keystore.p12 -inkey key.pem -in cert.pem
```

View the content of the PKCS#12 key store via:

```
openssl pkcs12 -info -in keystore.p12
```

When sending over your certificate to the AGIV Security Team, you will most likely have to deliver your certificate in DER format. Convert a certificate from PEM to DER via:

```
openssl x509 -in cert.pem -out cert.der -outform DER
```

4.1.2. Loading a key store

A PKCS#12 key store can be loaded as follows:

```
import java.io.InputStream;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.util.Enumeration;
import java.security.cert.X509Certificate;
import java.security.PrivateKey;

InputStream pkcs12InputStream =
    new FileInputStream("/path/to/your/keystore.p12");
KeyStore keyStore = KeyStore.getInstance("PKCS12", "SunJSSE");
keyStore.load(pkcs12InputStream, "your_password".toCharArray());

Enumeration<String> aliases = keyStore.aliases();
String alias = aliases.nextElement();

X509Certificate certificate = (X509Certificate) keyStore.getCertificate(alias);
PrivateKey privateKey = (PrivateKey) keyStore.getKey(alias,
    "your_password".toCharArray());
```

4.2. eID Certificates

The AGIV IP-STS also supports eID certificate credentials for testing purposes against the BETA IP-STS. This way you don't have to buy a certificate for testing the AGIV web services. Loading your eID authentication certificate and corresponding private key can be done using *Commons eID* [http://code.google.com/p/commons-eid/] .

First add the following dependency to your `pom.xml` under `<dependencies>` :

```
<dependency>
    <groupId>be.fedict.commons-eid</groupId>
    <artifactId>commons-eid-jca</artifactId>
    <version>0.4.0</version>
</dependency>
```

The following example demonstrates how to load your eID authentication certificate and corresponding private key:

```
import java.security.Security;
import be.fedict.commons.eid.jca.BeIDProvider;
import java.security.KeyStore;
```

```
import java.security.PrivateKey;
import java.security.cert.X509Certificate;

Security.addProvider(new BeIDProvider());
KeyStore keyStore = KeyStore.getInstance("BeID");
keyStore.load(null);
PrivateKey privateKey = (PrivateKey) keyStore.getKey("Authentication", null);
X509Certificate certificate = (X509Certificate) keyStore
    .getCertificate("Authentication");
```

5. Java EE 6

Enabling the AGIV Java Security framework within a full-blown Java EE 6 application server can be a challenge. Not only because JAX-WS is part of Java EE 6, but also because the embedded JAX-WS runtime stack comes with its own WS-* capabilities. The trick here is to have a *perfect alignment* of the WSS4J bundled within your enterprise application against the version of WSS4J provided by the application server. Hence it might be required to override the version of WSS4J that the AGIV Java Security framework will be using at run-time.

For example, for the JBoss AS 7.1.1.Final application server you have to change the WSS4J within your enterprise application to version 1.6.5. This can easily be done in Maven as follows.

```
<dependency>
    <groupId>org.apache.ws.security</groupId>
    <artifactId>wss4j</artifactId>
    <version>1.6.5</version>
</dependency>
```

Once you have the WSS4J run-time aligned with the one provided by the Java EE 6 application server, you can use the `@WebServiceRef` annotation to have the generated JAX-WS service instances injected into you EJB3 session beans.

Clients may experience delays because the framework needs to reacquire security tokens that are about to expire. In order to minimize these possible delays it is possible to explicity trigger the refreshment of all security tokens via:

```
agivSecurity.refreshSecurityTokens();
```

Where the `AGIVSecurity` instance is a `@Singleton` within the system. This could be done by some background process on a regular basis. The `refreshSecurityTokens` method returns the date on which the next security token in line will expire. This date can be used to program from example a Java EE 6 Timer Service.

A. AGIV Java Security Developer's Guide License

B. AGIV Java Security License

The AGIV Java Security source code has been released under the GNU LGPL version 3.0.

```
This is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License version 3.0 as published by the Free Software Foundation.
```

```
This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
```

```
You should have received a copy of the GNU Lesser General Public License along with this software; if not, see http://www.gnu.org/licenses/.
```

C. Revision history

Table C.1. Revision history

Date	Author	Description
31 Dec 2011	Frank Cornelis	Initial version.
11 Jan 2012	Frank Cornelis	STSListener.
19 Jan 2012	Frank Cornelis	JAX-WS runtimes.
29 Jan 2012	Frank Cornelis	X509 credentials.
31 Jan 2012	Frank Cornelis	Apache CXF 2.5.2.
3 Feb 2012	Frank Cornelis	Realms.
24 Mar 2012	Frank Cornelis	Optional WS-SecureConversation.
14 Apr 2012	Frank Cornelis	Java EE 6.
21 Apr 2012	Frank Cornelis	Tested new Apache CXF releases.
30 Apr 2012	Frank Cornelis	Tested new Java releases.
25 Aug 2012	Frank Cornelis	Tested new Java releases, JAX-WS RI, Metro, Apache CXF, Axis2.
26 Aug 2012	Frank Cornelis	Java EE 6 refreshing of security tokens.

Date	Author	Description
30 Apr 2013	Frank Cornelis	Tested new Java and JAX-WS runtime releases. Added service realms and locations.
25 May 2013	Frank Cornelis	Tested new Apache CXF releases. Manage certificate credentials section.