

# GT4 C WS A&A Developer's Guide

DRAFT

---

## GT4 C WS A&A Developer's Guide

### Introduction

This component contains mainly framework-level code and, as such, developing services and clients utilizing this component does in general involve either programmatically or declaratively driving the framework-level security code.

Now, what does this entail? On the programmatic side of things, it involves acquiring credentials, passing these credentials on to the framework, and setting various authentication- and protection-related flags, either in a descriptor or as properties on a stub object. On the declarative side, it involves setting up security descriptors, both client and service side, to prescribe the security policy used to drive the security framework code.

DRAFT

# Table of Contents

1. Before you begin .....	1
1. Feature summary .....	1
2. Tested platforms .....	1
3. Backward compatibility summary .....	1
4. Technology dependencies .....	1
5. Security considerations for C WS A&A .....	2
2. Usage scenarios .....	3
1. Delegation .....	3
2. Embedding Key Information in EPRs .....	3
3. Obtaining peer credentials on the server side .....	3
4. Obtaining peer credentials from message context on the client side .....	4
5. Using Authorization .....	5
6. Using Multiple Message Protection Schemes .....	5
3. Tutorials .....	7
4. Architecture and design overview .....	8
1. Transport Security .....	8
2. Message Level Security .....	8
5. APIs .....	11
1. Programming Model Overview .....	11
2. Component API .....	11
6. Services and WSDL .....	12
1. Secure Conversation Service .....	12
7. Framework-level Protocols .....	15
1. WS-Security .....	15
2. Transport (HTTPS) Security .....	15
I. Command-line tools .....	16
globus-credential-delegate .....	17
8. Domain-specific interface .....	18
1. Interface introduction .....	18
2. Syntax of the interface .....	19
9. Configuring .....	24
1. Configuration overview .....	24
2. Syntax of the interface .....	24
10. Environment variable interface .....	26
1. Environmental variables for C WS A&A .....	26
11. Debugging .....	27
1. Logging .....	27
12. Troubleshooting .....	28
1. Error Messages For C WS A&A .....	29
13. Related Documentation .....	34
Glossary .....	35

## List of Figures

4.1. The new certificate is signed by the owner, rather than a CA. ....	9
4.2. JAX-RPC handlers involved in security related message processing on a server. ....	10

DRAFT

## List of Tables

1. globus-credential-delegate options .....	17
8.1. Client side security properties .....	20
9.1. Configuring server side authentication and message/transport security .....	25
12.1. C WS A&A Errors .....	30

DRAFT

# Chapter 1. Before you begin

## 1. Feature summary

Features new in GT 4.2.0

None.

Other Supported Features

- Compliance with published IBM/Microsoft WS-Trust and WS-SecureConversation specifications
- Compliance with the Web Services Security 1.0 standard
- HTTPS support
- Message integrity protection.

Deprecated Features

- None.

## 2. Tested platforms

C WS A&A should work on any platform that supports J2SE 1.3.1 or higher.

Tested Platforms for C WS A&A

- Linux (Red Hat 7.3)
- Windows 2000
- Solaris 9

## 3. Backward compatibility summary

Since GT 4.0.x release, some incompatible changes have been made:

- Security Descriptors: The security descriptor schema has changed since GT 4.0.x and the descriptors from GT 4.0.x cannot be used as is.
- Secure Conversation port type: The WS Addressing version in Java WS Core has been updated and the secure conversation port type has changed to reflect this. Therefore, GT 4.0.x secure conversation clients are incompatible with GT 4.2.x servers and vice versa.

## 4. Technology dependencies

C WS A&A depends on the following GT components:

- C WS Core

GSI

C WS A&A depends on the following 3rd party software:

- [OpenSSL](http://www.openssl.org)<sup>1</sup>

## 5. Security considerations for C WS A&A

### 5.1. File permissions

The Java security code currently does not enforce secure permissions and, implicitly, file ownership requirements on any of the security related files, e.g. configuration and credential files. It is thus important that administrators ensure that the relevant files have correct permissions and ownership. Permissions should generally be as restrictive as possible, i.e. *private keys* should be readable only by the file owner and other files should be writable by owner only, and the files should generally be owned by the globus user (the requirements that the C code enforces are documented in [Configuring GSI](#)).

Also refer to [Section 5, “Known Problems”](#) for details on any other open issues.

---

<sup>1</sup> <http://www.openssl.org>

# Chapter 2. Usage scenarios

## 1. Delegation

There are two ways a client can delegate its credential to a service:

- using Delegation Service, and
- using GSI Secure Conversation.

A client can delegate using the [Delegation Service](#). This method is independent of the security scheme used and can be reused across multiple invocations of the client to multiple services (provided the services are in the same hosting environment as the Delegation Service). The link provided has details on client-side steps to delegate and service-side code to get the delegated credential.

GSI Secure Conversation has delegation built into the protocol. Delegation can be requested by setting the `GSIConstants.GSI_MODE` property on the Stub. If full or limited delegation is performed, the client credential can be obtained from the message context as follows:

```
Subject subject = (Subject) msgCtx.getProperty(Constants.PEER_SUBJECT);
```

The server can be configured such that container, service or client credentials are used for the operation invoked. For the client credentials to be used, the client should have delegated the credentials. Note that this is a server-side configuration. If `caller-identity` is chosen for the `run-as` configuration and the client's credentials have been successfully delegated, then the delegated credentials are associated with the current thread. The credentials in this case can be obtained as follows:

```
Subject subject = JaasSubject.getCurrentSubject();
```

## 2. Embedding Key Information in EPRs

GT provides an API to embed key information in an Endpoint Reference, as defined in the OGSA Basic Security Profile. The key information is embedded in the extensibility element of the EPR rather than the meta-data element as defined in the specification, since the toolkit uses older version of the WS Addressing specification.

This information would be useful to ascertain the expected identity of the service for authorizing the service or to get the public certificate of the resource to be used for encrypting the request to the service. The optional `usage` element in the embedded key information indicates the use of the embedded keys, either for signature or encryption.

The API is in class `org.globus.wsrfl.impl.security.util.EPRUtil`. The method to embed the certificates is called `insertCertificates` and the method to extract the key information is called `extractCertificates`. Please refer to [API documentation](#) for details on using the methods.

## 3. Obtaining peer credentials on the server side

The security handlers populate a Subject object with peer information. The following code can be used to access the peer credentials. Note that the message context needs to be associated with the thread.

```
import org.globus.wsrfl.security.SecurityManager;  
import javax.security.auth.Subject;
```

```
org.apache.axis.MessageContext mctx =
org.apache.axis.MessageContext.getCurrentContext();
SecurityManager manager = SecurityManager.getManager(mctx);
Subject subject = manager.getPeerSubject();
```

The following code snippet shows how the certificate chain can be extracted from the peer subject:

```
java.util.Set set =
subject.getPublicCredentials(X509Certificate[].class);
Iterator iterator = set.iterator();
while (iterator.hasNext()) {
X509Certificate[] certArray = (X509Certificate[]) iterator.next();
System.out.println("Cert array " + certArray.length);
}
```

To obtain the peer principal (for example, the Distinguished Name from X509 Certificate), the following code snippet can be used:

```
org.apache.axis.MessageContext mctx =
org.apache.axis.MessageContext.getCurrentContext();
SecurityManager manager = SecurityManager.getManager(mctx);
Principal principal = manager.getCallerPrincipal();
String caller = manager.getCaller();
```

If credentials were delegated as described above, private credentials are also populated:

```
java.util.Set set = subject.getPrivateCredentials();
```

## 4. Obtaining peer credentials from message context on the client side

- **GSI Secure Conversation:** With this mechanism, the peer credentials can be obtained once the handshake is completed:

```
import org.globus.wsrfl.impl.security.authentication.Constants;
import org.globus.wsrfl.impl.security.authentication.secureconv.service.S
import org.ietf.jgss.GSSContext;
import org.globus.gsi.gssapi.GSSContextants;

// Get current secure context from message context
SecurityContext secContext =
messageContext.getProperty(Constants.CONTEXT);
GSSContext gssContext = secContext.getContext();
Vector peerCerts =
```

```
gssContext.inquireByOid(GSSContants.X509_CERT_CHAIN);
```

- **GSI Secure Transport:** With this mechanism, the peer credentials can be obtained once the handshake is completed:

```
import org.ietf.jgss.GSSContext;
import org.globus.gsi.gssapi.GSSContants;
import org.globus.wsrfl.impl.security.authentication.Constants;

// Get current secure context from message context
GSSContext gssContext =
messageContext.getProperty(Constants.TRANSPORT_SECURITY_CONTEXT);
Vector peerCerts =
gssContext.inquireByOid(GSSContants.X509_CERT_CHAIN);
```

- **GSI Secure Message:** With this mechanism, the peer credentials can be obtained only when the response is received:

```
import org.globus.wsrfl.impl.security.authentication.Constants;

// Get peer subject from current message context
Subject subject =
(Subject) messageCtx.getProperty(Constants.PEER_SUBJECT);
Set peerCerts =
subject.getPublicCredentials(X509Certificate[].class);
```

## 5. Using Authorization

[describe what authz info there is for c ws aa]

## 6. Using Multiple Message Protection Schemes

Multiple message protection schemes can be used in a single invocation, although it is worth noting that this will cause a performance penalty.

For example, both Secure Transport and Secure Conversation can be done on the same invocation by using the following:

```
stub._setProperty(Constants.GSI_SEC_CONV, Constants.INTEGRITY);
stub._setProperty(Constants.GSI_TRANSPORT, Constants.PRIVACY);
```



### Note

These two mechanisms share a single property for authorization. There is a bug open to provide independent support: [Bug 4350](http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=4350)<sup>1</sup>

<sup>1</sup> [http://bugzilla.mcs.anl.gov/globus/show\\_bug.cgi?id=4350](http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=4350)

Similarly Secure Messages can be used in tandem with other message protection mechanisms.

DRAFT

# Chapter 3. Tutorials

There are no tutorials available at this time.

DRAFT

# Chapter 4. Architecture and design overview

## 1. Transport Security

The toolkit by default is deployed with our implementation of transport security, which is based on HTTP over SSL, also known as HTTPS, with modifications to path validation to enable X.509 *Proxy Certificate* support. In contrast to the GT3 version of the toolkit, the default transport security enabled in the toolkit does not support delegation of proxy certificates as part of the security handshake.

However, the underlying security libraries and handlers required for secure transport with delegation, also known as HTTPG, is still supported and shipped as part of the CoG library. The GT4 Java WS code base and configuration can be modified to use the HTTPG protocol as required.

Transport security is implemented by layering on top of the *GSISocket* class provided in JGlobus. This class deals with the security-related aspects of connection establishment as well as message protection. The socket interface serves as an abstraction layer that allows the HTTP protocol handling code to be unaware of the underlying security properties of the connection.

Container-level credentials are required and, irrespective of security settings on the service being accessed, these credentials are used for the handshake.

### 1.1. Server-Side Security

On the server-side, transport security is enabled by simply switching a non-secure socket implementation with the *GSISocket* implementation. In addition to this change, some code was added to propagate authentication information and message protection settings to the relevant security handlers, in particular the authorization and security policy handlers.

### 1.2. Client-Side Security

On the client-side, transport security is similarly enabled by switching a non-secure socket implementation with the *GSISocket* implementation and registering a protocol handler for HTTPS that uses the secure socket implementation. In practice, this means that any messages targeted at an HTTPS endpoint will, regardless of any stub properties, be authenticated and protected. It also means that any messages sent to an HTTP endpoint will not be secured, again regardless of any stub properties. Stub properties are only used to communicate the desired message protection level, i.e. either integrity only or integrity and privacy.

## 2. Message Level Security

### 2.1. Server Side Security

This section aims to describe the message flow and processing that occurs for a security-enabled service. The figure below shows the JAX-RPC handlers that are involved in security-related message processing on a server.

### Figure 4.1. The new certificate is signed by the owner, rather than a CA.



GT4 provides two mechanisms, **GSI Secure Conversation** and **GSI Secure Message** security, for authentication and secure communication.

- In the GSI Secure Conversation approach the client establishes a context with the server before sending any data. This context serves to authenticate the client identity to the server and to establish a shared secret using a collocated GSI Secure Conversation Service. Once the context establishment is complete, the client can securely invoke an operation on the service by signing or encrypting outgoing messages using the shared secret captured in the context.
- The GSI Secure Message approach differs in that no context is established before invoking an operation. The client simply uses existing keying material, such as an *X509 End Entity Certificate*, to secure messages and authenticate itself to the service.

Securing of messages in the GSI Secure Conversation approach, i.e. using a shared secret, requires less computational effort than using existing keying material in the GSI Secure Message approach. This allows the client to trade off the extra step of establishing a context to enable more computationally efficient messages protection once that context has been established.

## 2.2. Message Processing

When a message arrives from the client, the SOAP engine invokes several security-related handlers:

1. The first of these handlers, the **WS-Security handler**, searches the message for any WS-Security headers. From these headers, it extracts any keying material, which can be either in the form of an X509 certificate and associated certificate chain or a reference to a previously established secure conversation session. It also checks any signatures and/or decrypts elements in the SOAP body. The handler then populates a peer JAAS subject object with principals and any associated keying material whose veracity was ascertained during the signature checking or decryption step.
2. The next handler that gets invoked, the **security policy handler**, checks that incoming messages fulfill any security requirements the service may have. These requirements are specified, on a per-operation basis, as part of a security descriptor during service deployment. The security policy handler will also identify the correct JAAS subject to associate with the current thread of execution. Generally, this means choosing between the peer subject populated by the WS-Security handler, the subject associated with the hosting environment and the subject associated with the service itself. The actual association is done by the pivot handler, a non-security handler not shown in the figure that handles the details of delivering the message to the service.
3. The security policy handler is followed by an **authorization handler**. This handler verifies that the principal established by the WS-Security handler is authorized to invoke the service. The type of authorization that is performed is specified as part of a deployment descriptor. More information can be found in the [authorization framework documentation](#).

Once the message has passed the authorization handler, it is finally handed off to the actual service for processing (discounting any non-security-related handlers, which are outside the scope of this document).

Replies from the service back to the client are processed by two outbound handlers: the GSI Secure Conversation message handler and the GSI Secure Message handler. The GSI Secure Conversation message handler deals with encrypting and signing messages using a previously established security context, whereas the GSI Secure Message handler deals with messages by signing or encrypting the messages using X509 certificates.

The operations that are actually performed depend on the message properties associated with the message by the inbound handlers, i.e. outbound messages will have the same security attributes as inbound messages. That being said, a service has the option of modifying the message properties, if so desired. These handlers are identical to the client-side handlers described in the following section.

## 2.3. Client Side Security

This section describes the security-related message processing for Java-based GT4 clients. In contrast to the server side, where security is specified via deployment descriptors, client side security configuration is handled by the application. This means that a client-side application must explicitly pass information to the client-side handlers on what type of security to use. This is also true for the case of services acting as clients. The below figure shows the JAX-RPC handlers that are involved in security-related message processing on a server.

**Figure 4.2. JAX-RPC handlers involved in security related message processing on a server.**



## 2.4. Message Processing

The client-side application can specify the use of either the GSI Secure Conversation security approach or the GSI Secure Message security approach. It does this by setting a per-message property that is processed by the client-side security handlers.

There are three outbound client-side security handlers:

1. The **secure conversation service handler** is only operational if GSI Secure Conversation mode is in use. It establishes a security session with a secure conversation service collocated with the service with which the client aims to communicate. When the client sends the initial message to the service with a property indicating that session-based security is required, this handler intercepts the message and establishes a security session. It will also authorize the service by comparing the service's principal/subject obtained during session establishment with a value provided by the client application. Once the session has been established, the handler passes on the original message for further processing.
2. The next handler in the chain, the **secure message handler**, is only operational if GSI Secure Message mode is in use. It signs and/or encrypts messages using X.509 credentials.
3. The third outbound handler [fixme - is there a name?] is operational only if GSI Secure Conversation mode is in use. It handles signing and/or encryption of messages using a security session established by the first handler.

The client-side inbound handler (the WS-Security client handler) deals with verifying and decrypting any signed and/or encrypted incoming messages. In the case of the GSI Secure Message operation, it will also authorize the remote side in a similar fashion to the outbound secure conversation service handler.

# Chapter 5. APIs

## 1. Programming Model Overview

The security programming model differs between the client and server side. The client side model is programmatic in nature, i.e. security-related code is driven by making actual function calls, whereas the server-side model is declarative, i.e. security-related settings are declared in a security descriptor. For more information on the available client side calls see ????. More information about the security descriptor can be found in [Java WS A&A Security Descriptor Framework](#).

## 2. Component API

- Stable interfaces:
  - `org.globus.wsrp.security.Constants`
  - `org.globus.wsrp.security.SecureResource`
  - `org.globus.wsrp.security.SecurityManager`
  - `org.globus.wsrp.security.SecurityException`
- Less stable interfaces:
  - `org.globus.wsrp.impl.security.descriptor.ClientSecurityDescriptor`
  - `org.globus.wsrp.impl.security.descriptor.ResourceSecurityDescriptor`

Documentation for these interfaces can be found [here](#)<sup>1</sup>.

---

<sup>1</sup> [http://www.globus.org/api/javadoc-4.2.0/globus\\_java\\_ws\\_core](http://www.globus.org/api/javadoc-4.2.0/globus_java_ws_core)

# Chapter 6. Services and WSDL

## 1. Secure Conversation Service

### 1.1. Protocol overview

This service provides a mechanism for generating a security session, i.e the negotiation of a shared secret which may be used to secure a set of subsequent messages. It is based on the [WS-Trust](#)<sup>1</sup> and [WS-SecureConversation](#)<sup>2</sup> specifications.

### 1.2. Operations

- `RequestSecurityToken`: This operation initiates a new security session negotiation. Furthermore, since the actual schema for this message is not unambiguously defined by the specifications, this is the actual schema used:

```
<xs:element name='RequestSecurityToken'>
  <xs:complexType name='RequestSecurityTokenType'>
    <xs:sequence>
      <xs:element ref='wst:TokenType' />
      <xs:element ref='wst:RequestType' />
      <xs:element ref='wst:BinaryExchange' />
    </xs:sequence>
    <xs:attribute name='Context' type='xs:anyURI' />
  </xs:complexType>
</xs:element>

<xs:element name='RequestSecurityTokenResponse'>
  <xs:complexType name='RequestSecurityTokenResponseType'>
    <xs:sequence>
      <xs:element ref='wst:TokenType' />
      <xs:element ref='wst:RequestType' />
      <xs:element ref='wst:BinaryExchange' />
    </xs:sequence>
    <xs:attribute name='Context' type='xs:anyURI' />
  </xs:complexType>
</xs:element>
```

- `RequestSecurityTokenResponse`: This operation continues a security session negotiation. Furthermore, since the actual schema for this message is not unambiguously defined by the specifications, this is the actual schema used:

```
<xs:element name='RequestSecurityTokenResponse'>
  <xs:complexType name='RequestSecurityTokenResponseType'>
    <xs:sequence>
      <xs:element ref='wst:TokenType' />
      <xs:element ref='wst:RequestType' />
      <xs:element ref='wst:BinaryExchange' />
    </xs:sequence>
    <xs:attribute name='Context' type='xs:anyURI' />
  </xs:complexType>
```

<sup>1</sup> <http://www.ibm.com/developerworks/library/ws-trust/>

<sup>2</sup> <http://www-106.ibm.com/developerworks/library/ws-secon/>

```
</xs:complexType>
</xs:element>

<xs:element name='RequestSecurityTokenResponse'>
  <xs:complexType name='RequestSecurityTokenResponseType'>
    <xs:sequence>
      <xs:element ref='wst:TokenType' />
      <xs:element ref='wst:RequestType' />
      <xs:element ref='wst:BinaryExchange'
        minOccurs="0" />
      <xs:element ref='wsc:SecurityContextToken' />
    </xs:sequence>
    <xs:attribute name='Context' type='xs:anyURI' />
  </xs:complexType>
</xs:element>
```

In the above schema, the second RequestSecurityTokenResponse element refers to the final message in the exchange.

## 1.3. Resource properties

This service has no associated resource properties.

## 1.4. Faults

Both RequestSecurityToken and RequestSecurityTokenResponse throw the following faults:

- **ValueTypeNotSupportedFault:** This fault indicates that the value type attribute on the binary exchange token element is not supported by the service.
- **EncodingTypeNotSupportedFault:** This fault indicates that the encoding type attribute on the binary exchange token element is not supported by the service.
- **RequestTypeNotSupportedFault:** This fault indicates that the request type specified in the request type element is not supported by the service.
- **TokenTypeNotSupportedFault:** This fault indicates that the token type specified in the token type element is not supported by the service.
- **MalformedMessageFault:** This fault indicates that the message content received by the service does not conform to the expected content. This is necessary since the schema does not give a well defined content model.
- **BinaryExchangeFault:** This fault indicates that a failure occurred during the in the underlying security constant responsible for the session negotiation.
- **InvalidContextIdFault:** This fault indicates that the context id passed in the message is not valid within the context of this service or negotiation.

## 1.5. WSDL and Schema Definitions

- [WS-Trust WSDL](#)<sup>3</sup>

<sup>3</sup> <http://www-106.ibm.com/developerworks/library/specification/ws-trust/ws-trust.wsdl>

- [WS-Trust XSD](#)<sup>4</sup>
- [WS-SecureConversation XSD](#)<sup>5</sup>
- Secure Conversation WSDL [fixme - link]

DRAFT

---

<sup>4</sup> <http://www-106.ibm.com/developerworks/library/specification/ws-trust/ws-trust.xsd>

<sup>5</sup> <http://www-106.ibm.com/developerworks/library/specification/ws-secon/ws-secureconversation.xsd>

# Chapter 7. Framework-level Protocols

## 1. WS-Security

The framework implements the Web Services Security: SOAP Message Security<sup>1</sup>, Web Services Security: Username Token Profile<sup>2</sup> and Web Services Security: X.509 Token Profile<sup>3</sup> specifications.

## 2. Transport (HTTPS) Security

The transport security solution used by the framework consists of HTTP over SSL/TLS (HTTPS) using X.509 certificates. The path validation step has been augmented to support the Proxy Certificate Profile (RFC3820<sup>4</sup>).

---

<sup>1</sup> <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>

<sup>2</sup> <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf>

<sup>3</sup> <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf>

<sup>4</sup> <ftp://ftp.rfc-editor.org/in-notes/rfc3820.txt>

# Command-line tools

DRAFT

# Name

globus-credential-delegate -- Delegation client

globus-credential-delegate

# Tool description

Used to contact a Delegation Factory Service and store a delegated credential. A delegated credential is created and stored in a delegated credential WS-Resource, and the Endpoint Reference(EPR) of the credential is written out to a file for further use.

# Command syntax

globus-credential-delegate [options] <eprFilename>

**Table 1. globus-credential-delegate options**

[option1]	Enables anonymous authentication. Only supported with transport security or the GSI Secure Conversation authentication mechanism.
[option1]	Specifies the server's <i>certificate</i> file used for encryption. Only needed for the GSI Secure Message authentication mechanism.

# Chapter 8. Domain-specific interface

## 1. Interface introduction

Client-side security is set up by setting individual properties on the `javax.xml.rpc.Stub` object used for the web service method invocation or by setting properties on a client-side security descriptor object, which in turn is propagated to client-side security handlers by making it available as a stub object property. Here are examples of the two approaches:

- Setting a property on the stub:

```
// Create endpoint reference
EndpointReferenceType endpoint = new EndpointReferenceType();
// Set address of service
String counterAddr =
    "http://localhost:8080/wsrf/services/CounterService";
// Get handle to port
CounterPortType port =
    locator.getCounterPortTypePort(endpoint);
// set client authorization to self
((Stub)port)._setProperty(Constants.AUTHORIZATION,
    SelfAuthorization.getInstance());
```

- Setting properties using a client descriptor:

```
// Client security descriptor file
String CLIENT_DESC =
    "org/globus/wsrf/samples/counter/client/client-security-config.xml";
// Create endpoint reference
EndpointReferenceType endpoint = new EndpointReferenceType();
// Set address of service
String counterAddr =
    "http://localhost:8080/wsrf/services/CounterService";
// Get handle to port
CounterPortType port =
    locator.getCounterPortTypePort(endpoint);
//Set descriptor on Stub
((Stub)port)._setProperty(Constants.CLIENT_DESCRIPTOR_FILE, CLIENT_DESC);
```



### Note

If the client needs to use transport security, the following API must be used to register the Axis transport handler for https:

```
import org.globus.axis.util.Util;
static {
    Util.registerTransport();
}
```

## 2. Syntax of the interface

DRAFT

**Table 8.1. Client side security properties**

Number	Task	Stub Configuration
1.	Allows for configuration of credentials for authentication.	Property: <code>org.globus.axis.gsi.GSIConstants.GSI_CREDENTIALS</code> Value equals the Instance of <code>org.ietf.jgss.GSSCredential</code> .
2.	Allows for configuring client-side authorization.	Property: <code>org.globus.wsrf.security.Constants.AUTHORIZATION</code> Value equals the Instance of <code>org.globus.wsrf.security.authorization.Authorization</code> If GSI Secure Transport or GSI Secure Conversation is used, the value should be an instance of <code>org.globus.gsi.gssapi.auth.Authorization</code> . But this translation is done automatically by the toolkit.
3.	Enable GSI Secure Conversation with specified message protection level.	1. Property: <code>org.globus.wsrf.security.Constants.GSI_SEC_CONV</code> Values equal one of the following: <ul style="list-style-type: none"> <li>• <code>Constants.ENCRYPTION</code></li> <li>• <code>Constants.SIGNATURE</code></li> </ul> 2. Property: <code>org.globus.wsrf.security.Constants.GSI_SEC_CONV_SECREPLY_UNNECESSARY</code> If the value is set to <code>Boolean.TRUE</code> , the GSI Secure conversation protection is not required in the reply message. By default, if the request was secured with GSI Secure Conversation, the response is also required to have the same protection.           3. Property: You can set the SOAP Actor of the GSI signed/encrypted SOAP message by using the <code>gssActor</code> property. We recommend that you <i>not</i> do this unless you <i>really</i> know what you are doing.

4.	Sets the GSI delegation mode. <i>Used for GSI Secure Conversation only.</i> If limited or full delegation is chosen, then some form of client-side authorization needs to be done (i.e client-side authorization cannot be set to none).	<p>Property:</p> <pre>org.globus.axis.gsi.GSIConstants.GSI_MODE</pre> <p>Value equals one of following:</p> <ol style="list-style-type: none"> <li>1. <code>GSIConstants.GSI_MODE_NO_DELEG</code>: No delegation is performed.</li> <li>2. <code>GSIConstants.GSI_MODE_LIMITED_DELEG</code>: Limited delegation is performed.</li> <li>3. <code>GSIConstants.GSI_MODE_FULL_DELEG</code>: Full delegation is performed.</li> </ol>
5.	Enables GSI Secure Transport with some protection level.	<p>Property:</p> <pre>org.globus.gsi.GSIConstants.GSI_TRANSPORT</pre> <p>Values equal one of the following:</p> <ul style="list-style-type: none"> <li>• <code>Constants.ENCRYPTION</code></li> <li>• <code>Constants.SIGNATURE</code></li> </ul>
6.	Enables anonymous authentication. <i>This option only applies to GSI Secure Conversation and GSI Transport.</i>	<p>Property:</p> <pre>org.globus.wsrp.security.Constants.GSI_ANONYMOUS</pre> <p>Value equals one of following:</p> <ol style="list-style-type: none"> <li>1. <code>Boolean.FALSE</code>: Anonymous authentication is disabled.</li> <li>2. <code>Boolean.TRUE</code>: Anonymous authentication is enabled.</li> </ol>

7.	Enable GSI Secure Message with specified message protection level.	<p>1. Property:  <code>org.globus.wsrp.security.Constants.GSI_SEC_MSG</code></p> <p>Values equal one of the following:</p> <ul style="list-style-type: none"> <li>• <code>Constants.ENCRIPTION</code></li> <li>• <code>Constants.SIGNATURE</code></li> </ul> <p>2. Property:  <code>org.globus.wsrp.security.Constants.GSI_SEC_MSG_SECREPLY_UNNECESSARY</code></p> <p>If the value is set to <code>Boolean.TRUE</code>, the GSI Secure Message protection is not required in the reply message. By default, if the request was secured with GSI Secure Message, the response is also required to have the same protection.</p> <p>3. Property:  <code>org.globus.wsrp.security.Constants.GSI_SEC_MSG_SINGLECERT</code></p> <p>If the value is set to <code>Boolean.TRUE</code>, only a single certificate is used for the GSI Secure Message request. By default, the whole certificate chain is sent.</p> <p>4. Property:</p> <p>You can set the SOAP Actor of the signed message using the <code>x509Actor</code> property, but we do <i>not</i> recommend this unless you know what you are doing.</p>
8.	Enable WS-Security username/password authentication.	<p>Properties:</p> <p><code>org.globus.wsrp.security.Constants.USERNAME</code></p> <p>Value equals the username.</p> <p><code>org.globus.wsrp.security.Constants.PASSWORD</code></p> <p>Value equals the password.</p>

9.	Sets the credential that is used to encrypt the message (typically, the recipient's <i>public key</i> ). <i>Used for GSI Secure Message only.</i>	<p>Property:</p> <pre>org.globus.wsrfl.impl.security.authentication     .Constants.PEER_SUBJECT</pre> <p>Value equals the instance of <code>javax.security.auth.Subject</code>.</p> <p>The credential object needs to be wrapped in <code>org.globus.wsrfl.impl.security.authentication.encrypted</code> and added to the set of public credentials of the Subject object.</p> <p>For example, if <code>publicKeyFilename</code> was the file that had the recipient's public key:</p> <pre>Subject subject = new Subject(); X509Certificate serverCert =     CertUtil.loadCertificate(publicKeyFilename); EncryptionCredentials encryptionCreds =     new EncryptionCredentials(         new X509Certificate[] { serverCert }); subject.getPublicCredentials().add(encryptionCreds); stub._setProperty(Constants.PEER_SUBJECT, subject);</pre>
10.	Sets the trusted certificates location.	<p>Property:</p> <pre>org.globus.wsrfl.security.TRUSTED_CERTIFICATES</pre> <p>Value should be a comma-separated list of directories and file names.</p>
11.	Sets the SAML Authorization Assertion to embed in SOAP Header.	<p>Property:</p> <pre>org.globus.wsrfl.impl.security.authentication.Constants.SAML_AUTHZ_ASSERTION</pre> <p>Value should be an instance of <code>org.opensaml.SAMLAssertion</code>.</p>

Can  
con  
usin  
des

# Chapter 9. Configuring

## 1. Configuration overview

Configuration of service-side security settings can be achieved by using container or service security descriptor. Some of the security configuration, like the credential to use and trusted certificates location, can also be configured using CoG properties or rely on default location. **The preferred way is to provide these settings in a security descriptor.**

The next section provides details on the relevant properties. An overview of the syntax of security descriptors can be found in [Java WS A&A Security Descriptor Framework](#). Available CoG security properties can be found in [Chapter 2, Configuring](#)

## 2. Syntax of the interface

The following properties are relevant to authentication and message/transport security:

DRAFT

**Table 9.1. Configuring server side authentication and message/transport security**

Number	Task	Descriptor Configuration	Alternate Configuration
1	Credentials	<u>Container or service descriptor configuration</u> <sup>1</sup>	<ul style="list-style-type: none"> <li>• X509_USER_CERT or <u>CoG Configuration</u><sup>2</sup>: User certificate configuration</li> <li>• X509_USER_KEY or <u>CoG Configuration</u><sup>3</sup>: User key configuration</li> <li>• X509_USER_PROXY or <u>CoG Configuration</u><sup>4</sup>: User proxy configuration</li> </ul> <p>If no explicit configuration is found, the default proxy is read from /tmp/x509_up_&lt;uid&gt;.</p>
2	Trusted Certificates	<u>Container security descriptor configuration</u> <sup>5</sup>	<u>CoG Configuration</u> <sup>6</sup>
3	Limited proxy policy configuration	<u>Container or service descriptor configuration</u> <sup>7</sup>	None.
4	Replay Attack Window	<u>Container or service descriptor configuration</u> <sup>8</sup>	None.
5	Replay Attack Filter	<u>Container or service descriptor configuration</u> <sup>9</sup>	None.
6	Replay timer interval	<u>Container descriptor configuration</u> <sup>10</sup>	None.
7	Context timer interval	<u>Container descriptor configuration</u> <sup>11</sup>	None.

<sup>1</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-configCred>

<sup>2</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/common/javacog/admin/javacog-admin-configuring.html#javacog-admin-configuring-user-certificate>

<sup>3</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/common/javacog/admin/javacog-admin-configuring.html#javacog-admin-configuring-user-key>

<sup>4</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/common/javacog/admin/javacog-admin-configuring.html#javacog-admin-configuring-user-proxy>

<sup>6</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/common/javacog/admin/javacog-admin-configuring.html#javacog-admin-configuring-trusted-certs>

<sup>5</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-container-trusted>

<sup>7</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-rejectLimProxy>

<sup>8</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-replayAttack>

<sup>9</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-replayAttack>

<sup>10</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-container-replay>

<sup>11</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/security/wsaajava/wsaajava-secdesc.html#wsaajava-secdesc-container-context>

# Chapter 10. Environment variable interface

## 1. Environmental variables for C WS A&A

Refer to [Chapter 2, Configuring](#) for environment variables. Note that the above environment variable [fixme - not clear which envvar you mean] does not supersede any settings provided in security descriptors.

DRAFT

# Chapter 11. Debugging

Because C WS A&A is built on top of C WS Core, developer debugging is the same as described in [Chapter 8, Debugging](#).

For information about system administrator logs, see [Chapter 6, Debugging](#).

C WS Core also provides an API for CEDPs-compliant logging as described in [Section 2, “Logging”](#).

## 1. Logging

As of 4.2.0, the Globus Toolkit provides system administration logs that are [CEDPs best practices](#)<sup>1</sup> compliant.

To enable CEDPS logging, pass the `-log PATH` parameter to the `globus-wsc-container` program.

For more details on the CEDPS Logging format, including descriptions of reserved name-value pairs, see <http://cedps.net/index.php/LoggingBestPractices>:

### 1.1. Sample log file

The [sample log file](#)<sup>2</sup> contains many log entries for various scenarios in the C WS container.

---

<sup>1</sup> <http://cedps.net/index.php/LoggingBestPractices>

<sup>2</sup> <http://www.globus.org/toolkit/docs/4.2/4.2.0/common/cwscore/sample-container-log.txt>

# Chapter 12. Troubleshooting

For a list of common errors in GT, see [Error Codes](#).

DRAFT

# 1. Error Messages For C WS A&A

DRAFT

**Table 12.1. C WS A&A Errors**

Error Code	Definition
<pre>ERROR: Couldn't read user key: Bad passphrase key file location: /Users/bester/.globus/userkey.pem  globus_credential: Error reading user credential: Can't read credential's private key from PEM OpenSSL Error: pem_lib.c:423: in library: PEM routines, function PEM_do_header: bad decrypt OpenSSL Error: evp_enc.c:509: in library: digital envelope routines, function EVP_DecryptFinal: bad decrypt  Use -debug for further information.</pre>	Unable to decrypt private key
<pre>globus_gsi_gssapi: Error with gss credential handle globus_credential: Valid credentials could not be found in any of the possible locations specified by the credential search order. Valid credentials could not be found in any of the possible locations specified by the credential search order. Attempt 1 globus_credential: Error reading host credential globus_sysconfig: Error with certificate filename globus_sysconfig: Error with certificate filename globus_sysconfig: File is not owned by current user: /etc/grid-security/hostcert.pem is not owned by current user Attempt 2 globus_credential: Error reading proxy credential globus_sysconfig: Could not find a valid proxy certificate file location globus_sysconfig: Error with key filename globus_sysconfig: File does not exist: /tmp/x509up_u501 is not a valid file Attempt 3 globus_credential: Error reading user credential globus_credential: Key is password protected: GSI does not currently support password protected private keys. OpenSSL Error: pem_lib.c:401: in library: PEM routines, function PEM_do_header: bad password read</pre>	No user proxy could be found
<pre>globus_gsi_gssapi: Error with GSI credential globus_gsi_gssapi: Error with gss credential handle globus_credential: Error with credential: The proxy credential: /tmp/x509up_u1499     with subject: /DC=org/DC=example/DC=grid/OU=People/CN=Joe User/CN=1235439010     expired 44 minutes ago.</pre>	Proxy has expired.

Error Code	Definition
<p>globus_xio: The GSI XIO driver failed to establish a secure connection. The failure occurred during a handshake read.</p> <p>globus_xio: An end of file occurred</p>	<p>Communication disrupted during SSL handshake</p>
<p>globus_gsi_gssapi: Unable to verify remote side's credentials</p> <p>globus_gsi_gssapi: Unable to verify remote side's credentials: Couldn't verify the remote certificate</p> <p>OpenSSL Error: s3_pkt.c:1052: in library: SSL routines, function SSL3_READ_BYTES: sslv3 alert bad certificate SSL alert number 42</p>	<p>Unable to verify remote certificate. Often a clock-synchronization problem where the service clock is behind that of the client.</p>
<p>OpenSSL Error: s3_clnt.c:894: in library: SSL routines, function SSL3_GET_SERVER_CERTIFICATE: certificate verify failed</p> <p>globus_gsi_callback_module: Could not verify credential</p> <p>globus_gsi_callback_module: The certificate is not yet valid: Cert with subject: /DC=org/DC=example/DC=grid/OU=People/CN=Joe User/CN=464555355 is not yet valid- check clock skew between hosts.</p>	<p>Unable to verify remote certificate. Often a clock-synchronization problem where the client clock is behind that of the service.</p>

Error Code	Definition
<pre> globus_gsi_callback_module: Error with signing policy globus_sysconfig: Error getting signing policy file globus_sysconfig: File does not exist: /etc/grid-security/certificates/2b0e42b2.signing_policy is not a valid file </pre>	<p>The service's certificate is not trusted by the client</p>
<pre> globus_gsi_callback_module: Could not verify credential globus_gsi_callback_module: Error with signing policy globus_gsi_callback_module: Error in OLD GAA code: CA policy violation: &lt;no reason given&gt; </pre>	<p>Service certificate is not trusted because the CA signing policy does not trust the CA to sign the subject name of the certificate.</p>
<pre> Error: globus_soap_message_module: SOAP Fault Fault code: Client Fault string: globus_handler_ws_secure_message: Server Request handling failed globus_handler_ws_secure_message: Failed to verify the message: Unable to get Security header element from message attributes. </pre>	<p>The client sent a request to a service which message security without properly invoking the security handlers</p>

<b>Error Code</b>	<b>Definition</b>
<p>Error: globus_soap_message_module: SOAP Fault Fault code: Client Fault string: globus_soap_message_module: Loaded message handlers do not understand required header element: {http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd}Security</p>	<p>The client sent a request protected with message-level security but the server did not understand the required security headers</p>

# Chapter 13. Related Documentation

See [Section 9, “Associated standards for C WS A&A”](#) for a list of associated standards.

DRAFT

# Glossary

## C

certificate A public key plus information about the certificate owner bound together by the digital signature of a CA. In the case of a CA certificate, the certificate is self signed, i.e. it was signed using its own private key.

## P

private key The private part of a key pair. Depending on the type of certificate the key corresponds to it may typically be found in `$HOME/.globus/userkey.pem` (for user certificates), `/etc/grid-security/hostkey.pem` (for host certificates) or `/etc/grid-security/<service>/<service>key.pem` (for service certificates).

For more information on possible private key locations see [this](#).

proxy certificate A short lived certificate issued using a EEC. A proxy certificate typically has the same effective subject as the EEC that issued it and can thus be used in its place. GSI uses proxy certificates for single sign on and delegation of rights to other entities.

For more information about types of proxy certificates and their compatibility in different versions of GT, see <http://dev.globus.org/wiki/Security/ProxyCertTypes>.

public key The public part of a key pair used for cryptographic operations (e.g. signing, encrypting).